9th Workshop on Evaluation and Usability of Programming Languages and Tools

PLATEAU 2018, November 5, 2018,
Boston, Massachusetts, USA

Edited by
Titus Barik
Joshua Sunshine
Sarah Chasins

Editors

Titus Barik
Microsoft
titus.barik@microsoft.com

Joshua Sunshine
Carnegie Mellon University
sunshine@cs.cmu.edu

Sarah Chasins
University of California, Berkeley
schasins@berkeley.edu

ACM Classification 2012
Software and its engineering → Software notations and tools, Human-centered computing → Human computer interaction (HCI)

ISBN 978-3-95977-091-0

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-091-0.

Publication date
January, 2019

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/OASIcs.PLATEAU.2018

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 2190-6807

http://www.dagstuhl.de/oasics

Contents

Papers

9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018).
Editors: Titus Barik, Joshua Sunshine, and Sarah Chasins
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

Programming languages exist to enable programmers to develop software effectively. But programmer efficiency depends on the usability of the languages and tools with which they develop software. The aim of the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) is to discuss methods, metrics, and techniques for evaluating the usability of languages and language tools. The supposed benefits of such languages and tools cover a large space, including making programs easier to read, write, and maintain; allowing programmers to write more flexible and powerful programs; and restricting programs to make them more safe and secure.

The 9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018) was held on November 5, 2018 in Boston, Massachusetts, USA, and collocated with SPLASH 2018. The workshop gathered the intersection of researchers in the programming language, programming tool, and human-computer interaction communities to share their research and discuss the future of evaluation and usability of programming languages and tools.

It is our pleasure to present this year's proceedings. We are happy to report that we received six paper submissions, and that all papers were accepted after a thorough review process with at least two expert reviewers per paper. Together, these papers demonstrate the remarkable scope and applicability of the workshop, with topics that include software development techniques, software evolution, programming by example, and empirical studies in human-computer interaction.

Our thanks go to the authors, reviewers, speakers, and attendees, without whom this workshop would not have been possible.

Titus Barik, Joshua Sunshine, and Sarah Chasins
PLATEAU 2018 Co-Chairs

9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018).
Editors: Titus Barik, Joshua Sunshine, and Sarah Chasins

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee

Kelly Blincoe
University of Auckland

Ravi Chugh
University of Chicago

Luke Church
University of Cambridge

Loris D'Antoni
University of Wisconsin–Madison

Elena Glassman
University of California, Berkeley

Austin Henley
University of Tennessee

Felienne Hermans
Leiden University

Brittany Johnson
University of Massachusetts

Manuel Maarek
Heriot-Watt University

Éric Tanter
University of Chile

9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018).
Editors: Titus Barik, Joshua Sunshine, and Sarah Chasins

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A Randomized Controlled Trial on the Impact of Polyglot Programming in a Database Context

Phillip Merlin Uesbeck
University of Nevada, Las Vegas
Las Vegas, NV, USA
uesbeck@unlv.nevada.edu
https://orcid.org/0000-0001-8182-9942

Andreas Stefik
University of Nevada, Las Vegas
Las Vegas, NV, USA
stefika@gmail.com

Abstract

Using more than one programming language in the same project is common practice. Often, additional languages might be introduced to projects to solve specific issues. While the practice is common, it is unclear whether it has an impact on developer productivity. In this paper, we present a pilot study investigating what happens when programmers switch between programming languages. The experiment is a repeated measures double-blind randomized controlled trial with 3 groups with various kinds of code switching in a database context. Results provide a rigorous testing methodology that can be replicated by us or others and a theoretical backing for why these effects might exist from the linguistics literature.

2012 ACM Subject Classification Software and its engineering → Software development techniques

Keywords and phrases human-factors, randomized controlled trial, polyglot programming

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2018.1

Funding This work was partially funded by the NSF under grants 1644491, 1738259, and 1640131.

1 Introduction

Polyglot programming, using more than one computer language in the same programming context, is common. Evidence from Tomassetti and Torchiano suggests that 97% of open source projects on Github use two or more computer languages [19], with the number of languages per project averaging about 5 [11, 19]. Seeing how common the use of multiple programming languages is, the question of its productivity impact on developers at various levels becomes salient. An obvious research question might be, "Does polyglot programming have an impact on developer productivity, and if so, how large is it, what direction, and in what context?" Given that software is a $407.3 billion industry [1] and that the median salary for a software developer is $103,560 per year in the U.S. [12], productivity impacts are expensive at scale. This broad question, which we investigate one specific aspect of in this paper, guides our broader research direction.

One aspect of polyglot programming is repeated switching between languages, which we will call code switching. Using the study presented in this paper, we aim to evaluate whether code switching impacts developer productivity. Our running theory as to why this might be is guided by evidence-based research in the field of linguistics, which investigates

the phenomenon of code switching in natural language [8, 10, 21]. Research in that context has shown that there is a time cost to switching between the use of natural languages [13], which begs the question of whether a time cost is measurable for switching between computer languages. Given that neurological studies on program comprehension and natural language comprehension seem to use the same areas of the brain [14], such a hypothesis seems plausible.

The specific aspect of the broader research line we are investigating is in the common case of polyglot within a database programming context. Thus, we present here a small pilot study of a double-blind randomized controlled trial with six programming tasks. To do this, we created three different versions of a querying API, 1) one without polyglot (non), 2) one where raw SQL is embedded (polyglot), and 3) a hybrid approach of our own design where the embedded query is similar to the host language (hybrid). Our null-hypotheses is that there is no relationship between code switching and productivity in the context of database programming.

With guidance from this pilot study, our goal is to build up a research line in a systematic way, increasing the sample size as we go and replicate the work rigorously each time. Finally, many of the experiments our lab conducts are designed off of the work of Austin Bradford Hill [6], who led pioneering work in medicine on how to design experiments and which later helped lead to the CONSORT evidence standard. To our knowledge, the field of computer science does not have an evidence standard for empirical work, so we adapted CONSORT after discussions from Schloss Dagstuhl [15]. Put another way, while scholars sometimes disagree about what should be reported with empirical work, or how it is reported, we followed an existing evidence standard to provide an evidence-based structure on what we should and should not report.

The rest of this paper is going to be structured as follows: First, we will discuss related work. Then, we will go over the design of the experiment in section 3. Results will be shown in subsection 4 and then discussed in section 5, finally the paper will end in a conclusion in section 6.

2 Related Work

The benefits and drawbacks of polyglot programming on a human-factors level are under-explored in the scientific literature. The main argument in favor of polyglot programming is about *"[...] choosing the right tool for the job"* [2], a view that seems to drive the field of domain specific language research which proposes using specialized languages for better productivity and maintenance [20]. Claims have been made that the use of a more appropriate language for a task leads to better productivity and easier maintenance by reducing the lines of code of a project [7], but also that the need to learn more languages creates a strain on the developers and that introduction of more languages to a project can reduce the pool of developers able to maintain the project [7].

Programmer productivity is studied in a variety of aspects of programming. Studies range from programming language features such as syntax [16] and type systems [9], over API design [17] and the effect of errors [4] to studies trying to investigate the cognitive processes involved [14, 5]. These studies on APIs, syntax, or others provided guidance to this work, although there is little in the literature that contains measurements of polyglot programming itself.

Listing 1 Task 1 as presented to the participants

```
package library;

import library.*;

public class Task1 {

  /**
   * Please write this method to return a
       ↪ Table object containing
   * all columns for all entries with an id
       ↪ smaller than 32
   * and sorted from high salary to low salary
       ↪ .
   *
   * Table information:
   *
   * - prof -
   *
   * id (int) | firstname (String) | lastname
       ↪ (String) | salary (int)
   *
   *
   *
   * Use the technique shown to you in the
       ↪ samples given
   *
   */
  public Table query(Table prof) throws
       ↪ Exception {

    // Your Code here

    return null;
  }

}
```

Listing 2 Task 1 Solution polyglot

```
Query q = new Query();

q.Prepare("SELECT * FROM professors" +
" WHERE id < 32 ORDER BY salary DESC");

Table r = prof.Search(q);

return r;
```

Listing 3 Task 1 Solution non

```
Query q = new Query();

q.SortHighToLow("salary");
q.Filter(q.Where("id").LessThan(32));

Table r = prof.Search(q);

return r;
```

Listing 4 Task 1 Solution hybrid

```
Query q = new Query();

q.SortHighToLow("salary");
q.Filter("id < 32");

Table r = prof.Search(q);

return r;
```

3 Methods

Study Design. We conducted a double-blind repeated measures randomized controlled trial in which participants were randomly assigned to one of three experimental groups between April and May 2018. Each participant was asked to solve 6 programming tasks.

While we chose this particular design for our study, where participants used a host language with an embedded code, there are several other ways that it could have been designed and we are looking to expand testing to such designs at a later time. Under consideration was to have participants study one language in a task, then another in a second. We also considered testing smaller code samples to isolate certain aspects of switching, like just the one method where embedding occurs for polyglot. Ultimately we settled on a design that we thought was balanced. Since embedding SQL into a host language is common, it has some face validity as a common polyglot task and thus seemed like a reasonable place to investigate first.

Study Population & Setting. Eligible were participants over the age of 18 who had some programming experience. Recruitment occurred during class time via advertisement pamphlet at the University of Nevada, Las Vegas. The pamphlet contained the URL to the website used for the experiment which guided participants through the entire process of the experiment. On the website, participants give consent, fill out a survey, and then solve the programming tasks. A time limit of 45 minutes per task was given to limit the overall time commitment for the experiment.

Intervention. Participants were randomly assigned to one of three groups and received a different code sample depending on their group. Each group's code sample demonstrated enough code to let participants infer the solutions needed for the tasks. We designed the three different groups to require different amounts of language switching while writing database

queries in Java. The first group (polyglot) uses SQL strings as parameters in method calls to create a query (see listing 2 and 5), requiring switches. The second group (non-polyglot) had to create a query by building it from objects in a series of method calls in a more object-oriented approach (see listing 3 and 6), requiring no switches, and lastly, the hybrid group's approach mixes the use of parts of SQL-style strings and method calls, using SQL syntax for conditions and column names and method calls for the rest (see listing 4 and 7).

To solve the tasks, participants had to fill in the code provided to them in a way that satisfied all unit tests. If an incomplete solution was submitted, compilation output was displayed and work on the tasks continued until a successful solution was submitted or the time ran out. The participants had to solve 3 $SELECT$, 1 $UPDATE$, 1 $INSERT$, and 1 $JOIN$ task to cover a range of different uses of queries. While the $UPDATE$, $INSERT$, and $JOIN$ tasks were kept standard, the 3 $SELECT$ tasks varied in the complexity of their conditions and whether a $ORDER\ BY$ component was needed.

An example of what the tasks looked like can be seen in listing 1, which shows the empty first task of all three groups. The instructional comments describe the structure of the table object passed into the method. Possible solutions for each group to the first task can be seen in listings 2, 3, and 4.

Outcomes. The difference of the start time and the end time of a task decided the dependent variable time to solution. As a random factor, the platform also recorded the participants' self-reported experience in using databases.

Randomization. Randomized assignment to the groups was handled by the website and followed the covariate adaptive randomization approach [18]. The participants were assigned to a experience category based on their college year. Group assignment was then conducted randomly but balanced within experience categories.

Blinding. The assignment of participants to their group was done automatically and without intervention by the researchers. Since the experiment was done using the website, there was no direct interaction with the participants and therefore the proctors had fewer avenues to accidentally or intentionally bias them. The participants were not informed about which group they were assigned to or what the hypothesis of the study might be.

4 Results

Recruitment. We recruited 11 participants for this pilot study. Of the 11 participants, 5 identified themselves as female and 6 as male. On average, the participants were about 24 years old (M = 23.55, SD = 7.10). Six of the participants were sophomores, two were juniors, two were seniors, and one was a graduate student. Four of the participants reported that English is not their primary language. The polyglot group had 4 participants, the non-polyglot group had 3, and the hybrid group had 4. The demographics can also be found in table 1.

Baseline Data. All 11 of the participants completed all tasks. The data (all times in seconds) can be found in table 2. Figure 1 shows the average task completion times between the two groups. Figure 2 shows the task times by group in a boxplot.

Table 1 Demographics.

Metric	Polyglot	Non	Hybrid
N	4	3	4
DB Experience	25.00%	33.33%	25.00%
Female	50.00%	66.66%	0.00%
Age	23.50 (SD = 5.74)	20.33 (SD = 1.53)	26.00 (SD = 10.74)
Native	75.00%	66.66%	100.00%

Table 2 Times per task in seconds.

	Non			Polyglot			Hybrid			Total	
Task	N	mean	SD	N	mean	SD	N	mean	SD	mean	SD
Task 1	3	1371.67	545.68	4	1380.50	700.38	4	999.25	668.48	1239.45	614.05
Task 2	3	676.33	143.39	4	782.00	641.35	4	848.50	368.77	777.36	416.42
Task 3	3	1006.67	533.35	4	1870.25	419.55	4	1363.50	965.22	1450.45	722.35
Task 4	3	438.67	220.18	4	268.00	134.39	4	357.50	215.69	347.09	184.75
Task 5	3	154.33	39.95	4	702.50	945.13	4	258.75	107.21	391.64	578.24
Task 6	3	663.00	412.94	4	787.00	437.08	4	337.25	47.33	589.64	367.10
Total	18	718.44	507.60	24	965.04	751.49	24	694.12	615.21	799.27	645.89

Analysis. To analyze the results, we ran a mixed designs repeated measures ANOVA using the R programming language with respect to time to solution, using task as a within-subjects variable and group and database experience as between-subjects variables. Sphericity was tested using Mauchly's test for Sphericity, which shows that the assumption of sphericity was violated for the variable task. Following reported numbers are reported with Greenhouse-Geisser corrections taken into account.

There is a significant effect at $p < 0.5$ for the within-subjects variable task, $F(5, 25) = 7.065$, $p < 0.001$ ($\eta_p^2 = 0.479$), but no significant effect for group, $F(2, 5) = 0.889$, $p = 0.467$ ($\eta_p^2 = 0.110$) or database experience, $F(1, 5) = 0.973$, $p = 0.369$ ($\eta_p^2 = 0.064$). None of the interaction effects are significant. To test the differences between the tasks in more detail, we ran a Bonferroni test. There are significant differences between task 1 and 4 (p = 0.011), 1 and 5 (p = 0.041), 3 and 4 (p = 0.009), 3 and 5 (p = 0.008), and 3 and 6 (p = 0.035).

Listing 5 Solution 6 polyglot

```
Query q = new Query();
q.Prepare("SELECT id,
    ↪ firstname, lastname,
    ↪ clubname "+
  "FROM students JOIN clubmap
      ↪ ON students.id =
      ↪ clubmap.studentid");
Table r = students.Search(q,
    ↪ clubmap);
return r;
```

Listing 6 Solution 6 non

```
Query q = new Query();
q.AddField("id")
  .AddField("firstname")
  .AddField("lastname")
  .AddField("clubname");

q.Combine(students, "id",
    ↪ clubmap, "studentid");
Table r = students.Search(q);
return r;
```

Listing 7 Solution 6 hybrid

```
Query q = new Query();
q.AddFields("id, firstname,
    ↪ lastname, clubname");

q.Combine(students, "id",
    ↪ clubmap, "studentid");

Table r = students.Search(q);

return r;
```

5 Discussion

It is crucial to state that we found no significant differences overall in our pilot. This is what we would expect with a sample size of 11 unless polyglot had a very large effect size. We observed that $\eta_p^2 = 0.110$ for the group task, which means the polyglot effect explained about 11% of the variance. This needs to be confirmed or refuted at scale.

Besides the effect, which gives us clues toward what we might see in replications, we think observations about the individual tasks are interesting. Note that the non-polyglot group performed 3.50% slower than hybrid group across all tasks and that the polyglot group was

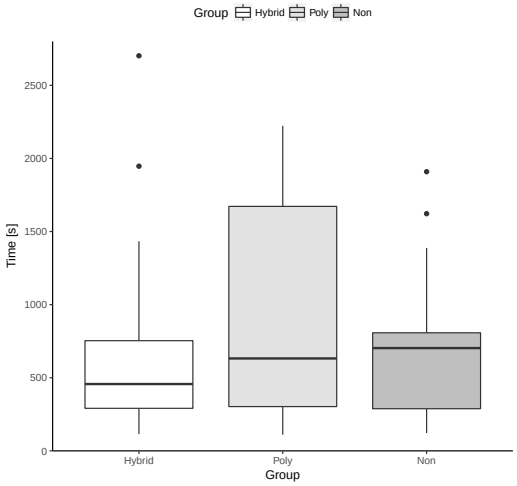

Figure 1 Boxplot of results between the groups.

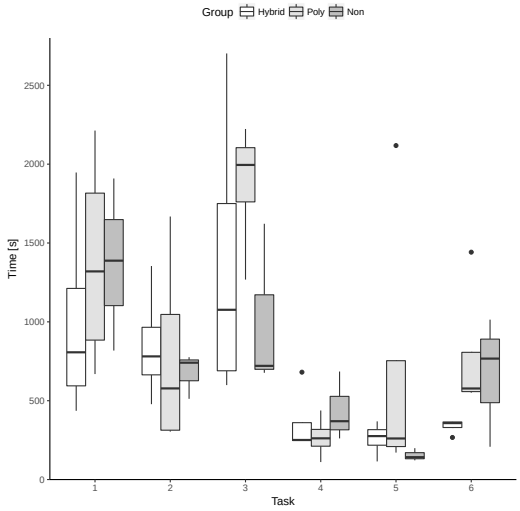

Figure 2 Boxplot of differences in time by task.

39.0% slower than hybrid. For some tasks, the effects were more pronounced. Consider task 6, where the non-polyglot group performed 96.6% slower than the hybrid group and the fully polyglot group was 133.4% slower than hybrid. This task is shown in listings 5, 6, and 7. We highlight this task not because of its result, but because the differences in the code are quite small. Given the large differences in human performance, this surprised us.

We think these observations teach us two lessons. First, polyglot programming is not a simple concept. Claims made in the literature about the practice being good or bad, without corresponding data, should be re-considered empirically. For example, after reading the linguistics literature on code switching, while we hypothesized the non-polyglot group might do very well, our polyglot hybrid seemed to have about the same impact. The study presented in this article is only a first step towards investigating the issue and we hope that the experimental design we presented can be a building block in future study of the impact of polyglot programming. Secondly, the polyglot group did quite poorly, despite the fact that we tested a commonly known case of polyglot programming. This might indicate, if our results are confirmed at scale and under more conditions, that a conditional polyglotism might be reasonable in programming languages. As the polyglot group uses two distinct languages and the hybrid group mixes languages while the last group stays within the same language, it appears that the distance from the host language could have an impact on results. This might suggest that switching between semantically and syntactically similar languages might be easier than switching between different ones.

Differences between tasks overall are explained by the differences in difficulty of the tasks and a learning effect. Task 3, presents a spike in times as it was the conceptually hardest task. It required participants to create a complex logical expression, compared to the ones for task 1 and 2, while most other tasks are what could be considered standard versions of their respective database commands.

Limitations. Every empirical study has limitations and ours is no exception. The most obvious to our pilot study is that it is a low sample size experiment on student programmers. We certainly make no claims about generalization, as this is not the point of a pilot study.

That said, it is important to discuss limitations not just in this sense, because generalization can mean many things, but also what limits must be overcome to get a generalized grasp of polyglot in practice.

First, we think to really understand polyglot in the wild, we would need a combination of measures to sort out the effects. For example, non-native English speakers writing code in English might have different results that need to be considered. On this same line of thinking, children learning to program, with various levels of understanding of their native tongue, would likely have different impacts as well. Similarly, professional programmers, or perhaps more specifically experts in polyglot programming in databases, might have different effects still. From our perspective, we think it is important to test all of these different kinds of people to sort out the facts over time.

Second, while we used a randomized controlled trial in a lab setting because our hybrid approach does not exist in the field, it is important to recognize that effects from a lab setting may not match those in practice, although in programming languages they sometimes do. Just as one example, studies on syntax [16] seem to provide similar results to those on compiler errors in the field [3], which Bradford-Hill would call "coherence." That said, there is no panacea in empirical work. Rigorous data gathering over time, through multiple techniques, is often what settles difficult research questions.

6 Conclusion

In this paper, we described a pilot experiment on the impact of code switching on software development productivity, which was motivated by the prevalence of the practice in the field. Findings in linguistic research suggest that there is a time cost to switching between natural languages, but to our knowledge this is the first randomized controlled trial on the topic for programming languages. We conducted a pilot study with three groups, exploring alternative designs for database programming, including polyglot, non-polyglot, and a hybrid approach. Our study was a small pilot designed to evaluate our methodology, so the results are not conclusive. That said, they do appear to provide a hint that the syntactic and semantic distance between embedded languages, amongst other factors, could impact human productivity in practice.

References

1 Gartner Says Worldwide Software Market Grew 4.8 Percent in 2013. `https://www.gartner.com/newsroom/id/2696317`. Accessed: 2018-06-02.

2 Polyglot Programming. `http://nealford.com/memeagora/2006/12/05/Polyglot_Programming.html`. Accessed: 2018-04-25.

3 Amjad Altadmri and Neil C.C. Brown. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 522–527, New York, NY, USA, 2015. ACM. `doi:10.1145/2676723.2677258`.

4 Brett A Becker. A new metric to quantify repeated compiler errors for novice programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 296–301. ACM, 2016.

5 Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*, pages 255–265. IEEE, 2015.

6 Richard Doll and A Bradford Hill. Smoking and carcinoma of the lung. *British medical journal*, 2(4682):739, 1950.

7 Hans-Christian Fjeldberg. *Polyglot programming. a business perspective.* PhD thesis, Master thesis, Norwegian University of Science and Technology, 2008.

8 Roberto R Heredia and Jeanette Altarriba. Bilingual language mixing: Why do bilinguals code-switch? *Current Directions in Psychological Science*, 10(5):164–168, 2001.

9 Michael Hoppe and Stefan Hanenberg. Do developers benefit from generic types?: an empirical comparison of generic and raw types in java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 457–474. ACM, 2013. doi:10.1145/2509136.2509528.

10 Ping Li. Spoken word recognition of code-switched words by Chinese–English bilinguals. *Journal of memory and language*, 35(6):757–774, 1996.

11 Philip Mayer and Alexander Bauer. An empirical analysis of the utilization of multiple programming languages in open source projects. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, page 4. ACM, 2015.

12 Bureau of Labor Statistics. Software Developers - Summary. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm. Accessed: 2018-04-20.

13 Daniel J Olson. Bilingual language switching costs in auditory comprehension. *Language, Cognition and Neuroscience*, 32(4):494–513, 2017.

14 Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 140–150. ACM, 2017.

15 Andreas Stefik, Bonita Sharif, Brad. A. Myers, and Stefan Hanenberg. Evidence About Programmers for Programming Language Design (Dagstuhl Seminar 18061). *Dagstuhl Reports*, 8(2):1–25, 2018. doi:10.4230/DagRep.8.2.1.

16 Andreas Stefik and Susanna Siebert. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.*, 13(4):19:1–19:40, November 2013.

17 Jeffrey Stylos and Brad A Myers. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 105–112. ACM, 2008.

18 KP Suresh. An overview of randomization techniques: an unbiased assessment of outcome in clinical research. *Journal of human reproductive sciences*, 4(1):8, 2011.

19 Federico Tomassetti and Marco Torchiano. An empirical assessment of polyglot-ism in GitHub. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 17. ACM, 2014.

20 Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

21 W Quin Yow, Jessica SH Tan, and Suzanne Flynn. Code-switching as a marker of linguistic competence in bilingual children. *Bilingualism: Language and Cognition*, pages 1–16, 2017.

Understanding Java Usability by Mining GitHub Repositories

Mark J. Lemay
Boston University, Boston, MA, USA
lemay@bu.edu

Abstract

There is a need for better empirical methods in programming language design. This paper addresses that need by demonstrating how, by observing publicly available Java source code, we can infer usage and usability issues with the Java language. In this study, 1,746 GitHub projects were analyzed and some basic usage facts are reported.

2012 ACM Subject Classification Human-centered computing → Empirical studies in HCI

Keywords and phrases programming languages, usability, data mining

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2018.2

Acknowledgements Thanks to my advisor Hongwei Xi for the encouragement to publish this research, the anonymous reviews who provided immense constructive feedback and to Stephanie Savir for correcting numerous errors.

1 Introduction

What makes a good programming language? While nearly every programmer has an opinion on what makes a programming language good, finding objective answers to this question is hard. While theoretical studies, like those in type theory, are important for the future of programming, theoretical properties like type safety and powerful constructs like dependent types have made little impact on mainstream software engineering. Theory may be necessary for "good" programming languages, but it is clearly not sufficient.

Another approach to measuring the "goodness" of languages comes from user studies. These studies generally take real people and have them perform some specific task using the language technology in question. While this approach has significantly improved some aspects of the mainstream programming experience[2], and hinted at interesting ways to develop a language[16] the scope of user studies is necessarily limited.

This paper proposes another way to measure the quality of programming languages: by analyzing publicly available source code artifacts such as those available on GitHub[1]. This approach alleviates many of the problems with user studies: very large samples are possible, the contributors are more likely to be experienced developers and projects are frequently large and realistic[2]. However, the data mining approach brings about new issues. We cannot directly ask users about their experiences, so there must be additional interpretation. Are programmers avoiding some features they find confusing and error prone? Or are they using an inconvenient feature frequently because the language is forcing them to? Aside from

[1] `https://github.com/`, GitHub is popular site for open source projects based on the git version control system

[2] This study includes popular libraries like spring-boot, guava, selenium, jenkins, junit and projects from organizations such as Netflix, Oracle, Paypal, Facebook and Google.

this, the programming language features we are interested in analyzing maybe underutilized for reasons other than their inherent usability: there might be a lack of education or features might be used indirectly through libraries. For instance, when observing the looping constructs of Java we see that the `do while` loop is very unpopular. This may be because of a lack of awareness of the future, rather then its inherent awkwardness. While conducting this research I found obscure Java features I was unaware of. Underlying language paradigms can also drastically change the usability of a feature. For instance, Haskell has no inherent notion of state, so a primitive "while" construct would not make sense. Hopefully data mining can provide a vastly different perspective from usability studies and theory that can help independently inform programming language design.

In this paper I mined the 1,746 most popular Java projects from GitHub. From this sample we can conclude a number of basic but novel facts about Java language usage. These facts will then be used to draw conclusions about the usability of different Java features, and suggest pain points that future languages should address. Additionally this paper demonstrates a simple method for analyzing Java files through the Eclipse IDE's parser[3].

2 Methodology

Java is one of the most popular programming languages and it has a large ecosystem of projects that can be analyzed. This makes Java a good candidate for data mining[4]. In addition, the Eclipse IDE's Java parser allows very precise information to be drawn from even very malformed files. Java is a relatively conservative language and invests heavily in backwards compatibility, so projects using very old versions of Java can be analyzed with little ambiguity.

In this study, the top Java GitHub repositories determined by star count[5] were selected by the GitHub search API and downloaded using an archive link. The most popular projects where chosen to avoid the many forks and copies of projects, and because it is likely that popular projects are more widely used and maintained by experienced developers. Some projects were randomly skipped over because of pagination issues with the search API[6]. Every repository that was available had each of its Java files parsed by the Eclipse IDE's parser into a traversable AST with the parser's best guess at partial type information. Because the Eclipse parser is designed to work with malformed files, it avoids several the issues other data mining projects have suffered from. This includes needing to know how to build the project, needing to resolve the correct version of library dependencies, and needing to find the correct version of the Java run time and Java language version (which is often not disclosed by build tools). Feature usages were then queried and aggregated.

3 Results

1,746 projects containing a total of 614,816 `.java` files and 97,758,514 lines of code was analyzed. The average Java file is 159 lines long.

---

[3] `https://www.eclipse.org/jdt/core/`

[4] I spent several years as a Java developer so I was experienced in the nuances of the language and the ecosystem.

[5] At the time of the download the most popular project had 37432 stars. The least popular project had 52 Stars.

[6] Fewer than 2% of projects were skipped. More careful scripts could avoid most of this error, but there will always be potential issues pulling data that is changing in real time while also respecting GitHub's rate limit.

Table 1 Control flow constructs.

Construct	Count/File[a]	Count
Return	6.2	3,825,353
If	4.7	2,878,814
Throw	0.74	455,898
Try	0.72	442,698
Catch Clause	0.64	396,475
For	0.52	317,699
Enhanced For[b]	0.44	271,766
Break	0.38	230,681
While	0.18	111,966
Switch	0.11	72,995
Continue	0.078	48,136
Synchronized	0.061	37,436
Do While	0.016	9,948
Labeled	0.0072	4,415

[a] This assumes that the count is averaged over all files in the sample, it is very likely some features are clustered together in non uniform ways.

[b] Added in Java 5, this variant of for loop allows collections to be traversed by element without an index.

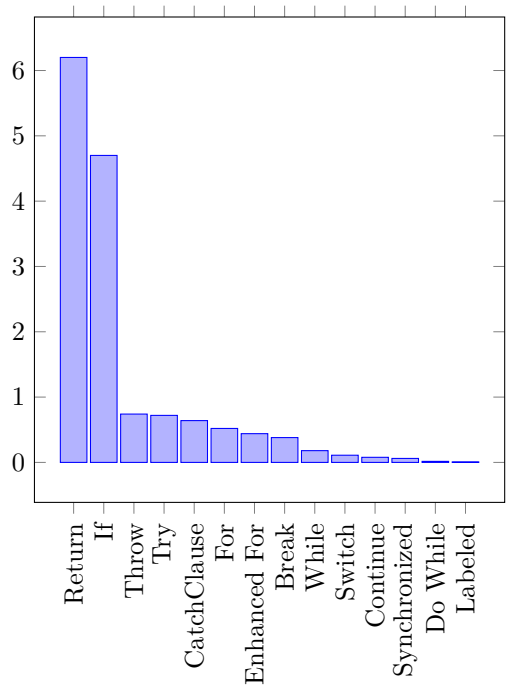

Figure 1 Control flow constructs, by Count/File

3.1 The Java Language

3.1.1 Control flow constructs

Java allows for several control flow constructs such as `for` loops, `switch` statements, `throw` and `catch` statements, and `return` statements. Table 1 shows the count of each construct from every `.java` file in the sample.

`return` is essentially required for writing Java functions, unsurprisingly it sees the heaviest usage.

`for` loops are by far the most popular looping construct. `while` loops are much less popular, though still used. Language authors should consider not including `do while` loops, since they seem to be avoided in practice. The obscure loop labeling construct that allows specific breaking of nested loops should be avoided in future languages.

It is interesting how much more popular the `if` statement is then the `switch` statement. Though, since `if` statements can be chained together to have switch like behavior, a direct comparison is questionable. This turns out not to be an issue, 82% of if statements have no else block, another 16% of `if`'s only have an `else` (with no directly nested `if`). `switch` statements eventually become more popular than `if else` chains, but usages of either is rare. This may mean that language authors should consider not including a `switch` construct, or instead include a more powerful pattern matching construct like those in functional languages like Haskell or Scala.

Table 2 Literal Usage.

Kind of Literal	Count/File	Count
Number	11.9	7,335,479
String	11.7	7,195,704
Null	3.4	2,098,983
Boolean	2.4	1,479,122
Character	0.35	214,443

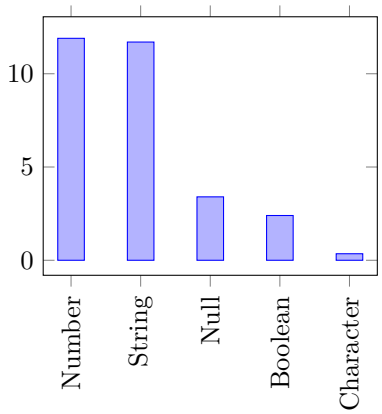

Figure 2 Literal Usage, by Count/File.

3.1.2 Literals

Literals are special syntactic constructs that a programmer may put in their code (for instance `"hello world"`, `'c'` , and `7`). Table 2 shows the count of each literal.

Developers rarely specify character literals. In fact, strings of length 1, occur 3 times as often as character literals. Language designers should consider not having special syntax for characters, instead relying on string syntax (as Python does).

The popular usage of `null` is interesting, and we will revisit this later.

3.1.3 Operators

Java does not allow operator overloading, so the 19 infix operators provided by the language are the only infix operators available. Were they well chosen? Table 3 shows the count of each operator.

Arithmetic and logic operators are very popular, but the bitwise operators are relatively unpopular. This is weak evidence that `x ^ y` might have been better used as the math power operator (instead of the rarely used XOR operator), though calls to `java.lang.Math::pow` occur less frequently.

3.1.4 Nulls

It turns out that the popularity of the `null` literal and the `==` operator are related.

In fact, over half of all equality checks are really `null` checks. This explains 59% of the null literals that occur in practice. Further inspection of `null` literals shows that 13% are used in method invocations, 13% are directly assigned or used in a declaration, and 7% are used in `return` statements. This weakly supports the popular idea that `null` references are a broken programming feature [8] and justifies special syntax for `null` checks in Kotlin, and the `Maybe` monad in Haskell.

3.2 The Java Standard Library

3.2.1 Most common method calls

Table 5 shows the most popular method call by name followed by the type that was most often resolved at the call site (methods with different signatures but the same name were counted the same for the sake of simplicity). The table shows that the collections libraries and string

Table 3 Infix operator usage.

Operator	Count/File	Count
x == y	1.98	1,216,367
x + y	1.89	1,164,466
x != y	1.39	855,485
x < y	0.70	430,412
x && y	0.68	415,305
x * y	0.58	355,789
x - y	0.56	342,409
x || y	0.38	231,446
x > y	0.31	189,792
x / y	0.24	149,703
x & y	0.15	92,747
x >= y	0.15	90,951
x <= y	0.12	74,888
x ^ y	0.08	46,641
x << y	0.06	35,205
x | y	0.04	26,638
x % y	0.04	23,412
x >> y	0.03	18,899
x >>> y	0.02	11,860

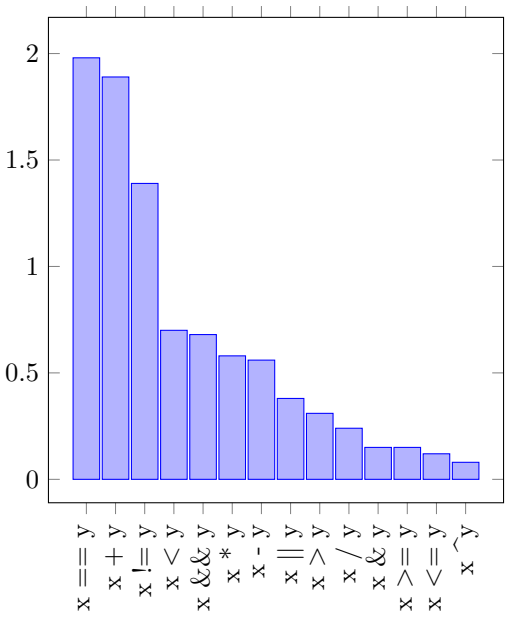

Figure 3 Infix operator usage, by Count/File.

Table 4 `null` checks.

Operator[a]	Count/File	Count
x != null	1.03	634,786
x == null	0.99	609,510
x == y	0.99	606,857
x != y	0.36	220,699

[a] `x == null` and `null == x` where counted the same.

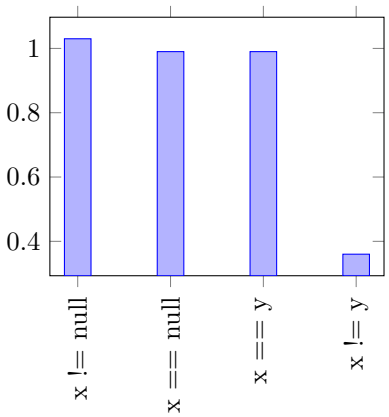

Figure 4 `null` checks, by Count/File.

Table 5 standard library calls.

Method	Count/File	Count	Most Common Example	Count/File	Count
append	0.67	412,426	java.lang.StringBuilder::append	0.57	347,825
get	0.52	320,393	java.util.List::get	0.17	105,124
add	0.48	294,697	java.util.List::add	0.28	173,803
put	0.34	208,357	java.util.Map::put	0.26	160,172
equals	0.3	187,119	java.lang.String::equals	0.21	130,928
size	0.27	163,659	java.util.List::size	0.16	100,976
toString	0.21	131,916	java.lang.StringBuilder::toString	0.08	48,729
println	0.18	110,867	java.io.PrintStream::println	0.15	90,464
length	0.11	70,160	java.lang.String::length	0.09	55,591
getName	0.11	68,646	java.lang.Class::getName	0.08	47,940
valueOf	0.1	64,421	java.lang.String::valueOf	0.03	21,129
hashCode	0.09	57,261	java.lang.String::hashCode	0.05	32,212
format	0.09	56,865	java.lang.String::format	0.08	46,710
isEmpty	0.08	51,011	java.util.List::isEmpty	0.03	19,546
contains	0.08	50,778	java.lang.String::contains	0.03	19,271
asList	0.08	47,444	java.util.Arrays::asList	0.08	47,432
getMessage	0.07	46,099	java.lang.Exception::getMessage	0.04	22,045
substring	0.06	38,717	java.lang.String::substring	0.06	38,071
next	0.06	36,884	java.util.Iterator::next	0.05	30,500
remove	0.06	35,878	java.util.Map::remove	0.01	7,849

operations make up a large fraction of method calls, and should be considered important for languages and standard libraries. Efficient string composition should be prioritized in future languages: optimizing string concatenation (with the `+` operator) would have made Java programs that use the appending function more readable. Almost every listed standard library call was more popular than `>>>`, the least popular infix operation.

4 Threats to Validity

There are some reasons to be concerned with this analysis

- Most software development is proprietary, and the open source projects on GitHub may be unrepresentative of non-open source projects.
- The most popular open source Java projects may not be representative of open source projects in general. For instance there was at least one satirical project in the sample[7].

[7] `https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition`

- Though it is frowned upon to check in generated code, it is still likely to happen in practice. Generated code could cause a bias in favor of constructs produced by the generation procedure.
- Projects that themselves try to parse Java (like the IntelliJ IDE[8]) will often include pathological test cases that contain extreme examples of Java syntax.
- It is also possible that some unknown bias was introduced though the Eclipse parser. Results should be compared against future versions of the parser.

5 Prior Work

There has been a large amount of work in mining software repositories[4]. While most of this work has focused on answering questions unrelated to language usability, there have been some studies worth mentioning.

5.1 Notable Java data mining projects

- In [11], Java projects from the open source repository SourceForge were mined to analyze 3rd party library usage and migration. This paper improves on their methodology by using the Eclipse parser, so projects can be analyzed in a reliable way without needing build information.
- [9] analyzed 22,730 artifacts in the maven package repository, to understand the landscape of the Java library ecosystem. Since projects in the maven system are expected to be used as libraries and require some basic level of quality to be accepted, this may not be a representative sample of Java projects in general.
- The Boa infrastructure[6] has the most similar methodology to this paper. It uses the Eclipse parser to analyze a snapshot of all Java projects with full git histories and uses Hadoop[9] to quickly mine large numbers of projects. The Boa infrastructure also includes the Boa programming language designed for non-expert computer users as in interface to the data mining infrastructure and Eclipse parser. The language and snapshots are made publicly available online[10]. In [7] the Boa infrastructure was used to analyze the adoption of Java features before Java 8. While Boa has some clear improvements over the methodology in this paper in terms of speed and sample size, only [7] dealt with Java usability directly. This paper does improve on a few details: Boa's latest publicly available snapshots are from 2015 at the time of this writing, while the projects in this paper were pulled in August 2018.

5.2 Other studies that address usability through data mining

There are several papers that have looked into usability of specific language features. There has been extensive research into usage of Java's exception handling mechanism[10, 13, 1, 15, 3]. Dikstra's skepticism of the `goto` construct[5] has been empirically tested with 384 C files from GitHub[12]. The usage of Scala's implicit parameter feature was analyzed in 120 GitHub projects to inform how to extend the feature[14].

[8] `https://github.com/JetBrains/intellij-community`
[9] `https://hadoop.apache.org/`
[10] `http://boa.cs.iastate.edu/boa`

5.3 Unpublished work from 2015

In unpublished work from 2015, I used a similar methodology to scan 1,970 GitHub repositories to understand adoption of Java 8. At the time only 14% of the projects had observably adopted Java 8 (by using one of Java 8 features). Of those projects, there was a noticeable decrease in single method anonymous classes, the Java 7 feature that most closely resembled the lambdas that appeared in Java 8. Lambdas were by far the most popular syntactic construct introduced by Java 8.

6 Future Work

There are a number of interesting directions to take this research

- Compare language usage across project types. There are many different uses for Java in practice: Java is heavily used in web development, Java was the primarily supported language for phone development under Android, and Java is used extensively for analytics with projects like Hadoop. It is now much easier to categorize project types since GitHub added "topics" in 2017[11]. Topics are tags that are generated automatically via machine learning[12] to project repositories and then curated by the project owner. This would offer a straightforward way to see how different kinds of projects use a language and its standard library differently.
- Reproduce the results of this paper under the Boa infrastructure. This would help increase confidence in this study as well as making the methodology more reproducible.
- Extend the analysis to different languages and paradigms. It would be interesting to see how Java usage compares to Python. It would be very interesting to see if a functional language like Haskell has similar usage.
- Run a similar analysis on popular libraries. Popular libraries like the Apache Commons[13] might give insight into features that should be incorporated into future standard libraries out of the box. An analysis like this could be very helpful for the library authors as well.
- Observe changes in usage over time. Since git keeps a record of a repositories history it would be interesting to see if some features become more or less popular over time. Do users upgrade to newer versions of the language and libraries? How quickly are new features adopted?
- Analyze the "real" types of literals such as strings. In Java strings are often used to to represent specific languages like regular expressions, SQL, or English. Analyzing the string literal usage would allow language designers to know when or if these languages should be made into separate first class languages, or should be handled in the standard library with string interpolation.
- More analysis should be done into the usage of `null` in practice. Does it very by project type? and has it become less popular over time? It would also be interesting to see how frequently the `notNull` annotation is used.

7 Conclusion

Data mining has the potential to inform many aspects of future language and library design; this paper barely scratches the surface. Hopefully, these techniques will suggest how future languages should be designed to make programming a more productive, safe, and enjoyable experience.

[11] `https://help.github.com/articles/about-topics/`
[12] `https://githubengineering.com/topics/`
[13] `https://commons.apache.org/`

References

1. Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K Roy, and Kevin A Schneider. How developers use exception handling in Java? In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 516–519. ACM, 2016.
2. Steven Clarke. Chapter 29. How Usable Are Your APIs? In Andy Oram and Gregn Wilson, editors, *Making software: What really works, and why we believe it*. O'Reilly Media, Inc., 2010.
3. Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. Unveiling exception handling bug hazards in Android based on GitHub and Google code issues. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 134–145. IEEE, 2015.
4. Valerio Cosentino, Javier Luis, and Jordi Cabot. Findings from GitHub: methods, datasets and limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 137–141. ACM, 2016.
5. Edsger W Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.
6. Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE'13, pages 422–431, 2013.
7. Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *Proceedings of the 36th International Conference on Software Engineering*, pages 779–790. ACM, 2014.
8. Tony Hoare. Null references: The billion dollar mistake. *Presentation at QCon London*, 298, 2009.
9. Vassilios Karakoidas, Dimitris Mitropoulos, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. Generating the blueprints of the Java ecosystem. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 510–513. IEEE Press, 2015.
10. Mary Beth Kery, Claire Le Goues, and Brad A Myers. Examining programmer practices for locally handling exceptions. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 484–487. IEEE, 2016.
11. Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1317–1324. ACM, 2011.
12. Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E Hassan. An empirical study of goto in C code from GitHub repositories. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 404–414. ACM, 2015.
13. Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of exception handling patterns in Java projects: An empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 500–503. ACM, 2016.
14. Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: foundations and applications of implicit function types. *Proceedings of the ACM on Programming Languages*, 2(POPL):42, 2017.
15. Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. Understanding the exception handling strategies of Java libraries: An empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 212–222. ACM, 2016.
16. Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4):19, 2013.

Programming by Example: Efficient, but Not "Helpful"

Mark Santolucito[1]
Yale University, USA
mark.santolucito@yale.edu

Drew Goldman
Roslyn High School, USA
dgoldman19@roslynschools.org

Allyson Weseley
Roslyn High School, USA
aweseley@roslynschools.org

Ruzica Piskac[2]
Yale University, USA
ruzica.pikac@yale.edu

Abstract

Programming by example (PBE) is a powerful programming paradigm based on example driven synthesis. Users can provide examples, and a tool automatically constructs a program that satisfies the examples. To investigate the impact of PBE on real-world users, we built a study around StriSynth, a tool for shell scripting by example, and recruited 27 working IT professionals to participate. In our study we asked the users to complete three tasks with StriSynth, and the same three tasks with PowerShell, a traditional scripting language. We found that, although our participants completed the tasks more quickly with StriSynth, they reported that they believed PowerShell to be a more helpful tool.

2012 ACM Subject Classification Software and its engineering → Programming by example

Keywords and phrases user study, scripting, programming by example

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2018.3

1 Introduction

Scripting languages, such as PowerShell and bash, help IT professionals to more efficiently complete tedious and repetitive tasks. Those tasks can include file manipulations and organizing data, where a simple error can destroy users' data. As an example, consider the disastrous attempt to remove all backup emacs files with the command `rm * ~`. Additionally, small errors in the scripts can lead to malicious behavior, such as data loss [24]. Scripts can be difficult for users to write by hand, requiring users to have extensive experience with regular expressions, programming and domain expertise in the scripting language of their choice. Depending on the application, a user may need to be able to write a very complicated regular expression for a relatively simple task. Furthermore, users may not have access to their scripting language of choice, depending on the operating system and software policies used by their employer.

[1] This research sponsored by NSF grants CCF-1302327, CCF-1553168 and CCF-1715387.
[2] This research sponsored by NSF grants CCF-1302327, CCF-1553168 and CCF-1715387.

For these reasons, many end-users search for help on online forums when they need to write a script [3, 2, 4]. When users seek help in writing a script on forums, they will often provide a few illustrative examples that convey the goal of the script. This observation was the basis of StriSynth [13], a research tool that was proposed to make scripting easier and more efficient by allowing users to program scripts by example. While scripting is a challenging task, especially for novice programmers, providing examples of the intended behavior is a more natural interface for scripting. StriSynth supports various types of functions, such as transformations, filters, partitions, and merging strings.

In this work, we explore how scripting by example, specifically with StriSynth, is received by the real-world target end-users. We designed a user study around StriSynth and recruited 27 IT professionals to participate in the study. In our study we asked users to complete three tasks with StriSynth, and the same three tasks with PowerShell, a traditional scripting language. When using StriSynth, users were statistically significantly faster at completing tasks as compared with PowerShell. However, in a post-study survey when users were asked which tool they perceived to be more "helpful", users statistically significantly reported that PowerShell, with the traditional scripting paradigm, was more helpful. This was counter-intuitive result, as we expected that a faster should be considered to be more helpful by users. While the formal methods community has largely taken efficiency of task completion to be an indicator of a good language design, we explore our results here that show this is in fact a more complex issue.

2 Background

Programming by example (PBE) [6, 23, 26, 7] is a form of program synthesis. It works by automatically generating programs that coincide with the given examples. In this way, the examples can be seen as an incomplete, but easily readable and understandable specification. However, even if the synthesized program satisfies all the provided examples, it might not correspond to user's intentions, due to this incompleteness in the specification. In this case, a user must provide further examples to the synthesis tool.

To address this issue, StriSynth was implemented as a live programming environment [5] for PBE. In this way, a synthesized script can be refined with every new provided example, and thus yields more interactive experience for the user. Interactive PBE allows end-users to provide a single example at a time, rather than guessing at the full example set that is necessary for synthesis.

In order to compare the PBE paradigm to more traditional scripting languages, we have chosen to use the tool StriSynth [13]. StriSynth is an existing tool for automating file manipulation tasks, in a similar style to Flash Fill's [1] synthesis of spreadsheet manipulations. While the use of scripting language such as sed, awk, Bash or PowerShell requires a certain level of expertise, many tasks can be easily described using natural language or through examples.

2.1 StriSynth example

To give some context for how StriSynth compares to traditional scripting language paradigms, we give an example task that can be easily completed with StriSynth. This task comes from a StackOverflow post, where the users discuss challenging regular expressions [3]. The user asked for a script that will create a link from every item in a directory. To better illustrate the goal of the script, the user provided two examples transformations:

```
Document1.docx      ⇨    <A HREF='Document1.docx'>Document1</A>
Document2.docx           <A HREF='Document2.docx'>Document2</A>
```

To accomplish this transformation, other users on the forum suggested a solution based on regular expressions in `sed`:

```
sed/\(^[a-zA-Z0-9]+\)\.\([a-z]+\)/\<a href\=\'\1\.\2\' \>\1\<\/a\>/g
```

While it was very easy for the user to express the goal of the script by providing examples, the resulting script is arguably less readable, even for such a simple problem. In contrast, to solve this problem in StriSynth, a user provides an example showing what a script should do:

```
> NEW
> "f.docx" ==> "<a href='f.docx'>a</a>"
> val F = TRANSFORM
```

The keyword `NEW` denotes the start for learning of a new script, after which the user provides an example of the scripts desired behavior. Based on the provided example, StriSynth learns a string transformer, and the user saves it with the next command. Every learned function can be saved using the command `val name = ...` which creates a reference, `name`, to the learned script. The user may then check how F works on different examples to confirm the learned function is correct.

```
> F("Document1.docx")
<A HREF='Document1.docx'>Document1</A>
> F("Document2.docx")
<A HREF='Document2.docx'>Document2</A>
```

We observe that the learned transformer F is a function that exactly does what the user asked initially. However, it only takes a single string as input, while the user wanted a script that operates on a list of strings. To extend the learned transformer to work over a list, the user can use the `as map` function.

```
> val finalScript = F as map
```

If a function G has a type signature $G : T_1 \to T_2$, then applying the postfix operator `as map` will result in G `as map` $: \text{List}(T_1) \to \text{List}(T_2)$. With `as map`, the user creates the final script which takes as input a list of file names and creates a list of HTML links.

Beyond the string transformation used above, StriSynth can also learn other types of functions from examples. StriSynth supports a *filter* function that takes a list of strings as input and removes some elements based on the filtering criterion. Similarly, StriSynth also supports learning a *partition* function takes as input a list of strings, and divides them into groups based on the partitioning criterion. Those groups are then returned as a list of lists of strings. This functions can be used in any way by the user, but are particularly useful for scripting tasks that require operations on certain types of files, or files matching some naming pattern.

In addition, StriSynth can learn a *reduce* function that merges the elements in a list into a single string. StriSynth's *split* function does the opposite: it returns a list of strings from the input string. These types of functions are especially useful for scripting tasks that apply operations to collections of files.

3 Methodology

A recent survey of the key challenges facing formal methods cites the need for more user studies, especially on real-world users [16]. To test the impact PBE on real users, we recruited 27 IT professionals, all of whom were 18 years of age or older. All materials for the study, as well as the raw data results from the study are available open source at `https://github.com/santolucito/StriSynthStudy`.

Our study design consisted of four stages:

1. A tutorial on both PowerShell and StriSynth that introduced the paradigm and syntax
2. Complete three scripting tasks (*Extract filenames* from a directory listing, *Move files* with `*.png` to `imgs/`, *Printing pdfs* from a list of various file types) in PowerShell
3. Complete the same three scripting tasks in StriSynth
4. A post-study survey

In the study, participants were told that they would be using the tools StriSynthA and StriSynthB instead of StriSynth and PowerShell to avoid bias from participants' prior experience. The participants were randomly split into two groups, group A and group B, where the two groups switched the order of steps 2 and 3 of the study to account for any potential bias in earlier exposure to the tasks. Group A completed the tasks with PowerShell first (N=12) and group B completed the tasks with StriSynth first (N=15). The entire study generally took each participant 50 minutes, and the study was conducted in-person with a researcher present. The scripting tasks were completed on the researcher's laptop, which was preloaded before each study with directories and files needed for the scripting tasks.

While each user was participating in the study, the researcher present recorded the overall time that was used to complete each task. Following the completion of the six tasks, each user was given a questionnaire. The questionnaire measured various responses: prior coding experience, perceived helpfulness of each program as a whole, and perceived helpfulness of each program for each specific task they completed.

4 Results

In this section we present the results of the user study described in Sec. 3. Overall, users completed the tasks more quickly when using StriSynth as opposed to PowerShell. This is good evidence that StriSynth is an efficient tool, especially as none of our users had used StriSynth before this study, while some already had experience with PowerShell. However, despite this concrete measure of efficiency for StriSynth, users said that they believe that PowerShell is a more helpful tool.

4.1 Time to complete the user study tasks

To estimate the usefulness of the programming by example tool StriSynth, we recorded the time it took for users to complete each task with both StriSynth and PowerShell. The results are shown in Fig. 1. In addition, Fig. 1 also contains standard error, depicted with line bars.

In the case of the first task (extracting filenames), from in Fig. 1 the standard error bars give us the intuition that true mean of the time it takes for overall population to complete this task using PowerShell is between 170 and 210 seconds. The smaller the standard error, the more likely is that we have achieved the exact, true value of the mean time, which it takes for the entire population of IT professionals to complete the tasks.

We can see in Fig. 1 that overall the users took less time to complete the tasks with StriSynth. However, our sample size was relatively small (N=27). Therefore, we wanted

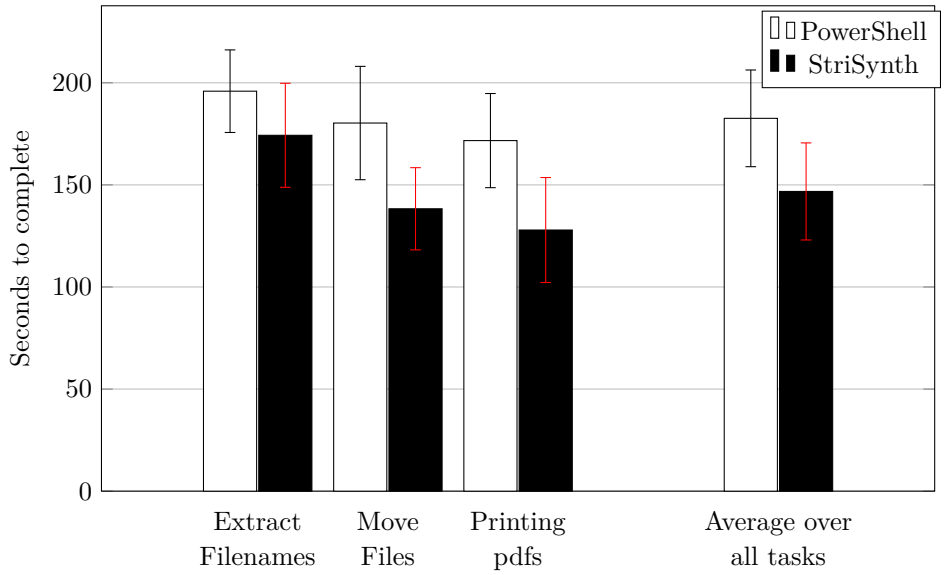

Figure 1 The amount of the time each task took, as well as the average time over all tasks for all users (N=27). The smaller bars indicate standard error.

to measure the confidence that our observations are reflective of the larger IT population beyond our small sample size. To do this we ran a paired sample t-test [30].

When running the paired sample t-test, we are checking the null hypothesis that the difference between the paired observations in the two samples is zero. Without going into the details of statistical methods, we need to compute the p-value. Any p-value of less than .05 is called *statistically significant*, indicating we have met a generally accepted threshold of confidence in our results [30].

By running these tests on our samples, we learn that a statistically significant difference was found in the *Move Files* ($p = .03$) and *Printing pdfs* ($p = .02$) tasks. The p-value of .03 means that, assuming StriSynth does *not* actually have any impact on time to complete the *Move Files* task, there is only a 3% chance that we could have observed the timing difference between StriSynth and PowerShell (or even some larger difference) presented Fig. 1. In other words, using StriSynth does actually have the impact on time to complete the task.

All together, our results support the claim that, for small scripting tasks of the type we presented to our users, PBE can be a more efficient programming paradigm.

4.2 Reported helpfulness

At the end of the study we asked users to report how "helpful" they found both StriSynth and PowerShell. At this point, users did not know how long they took to complete the tasks with each of the tools. Users were asked the rate the helpfulness only based on their experience of using the tools during the study. The exact questions asked were "The following program was helpful for scripting/completing *Extract Filenames*/etc...", and users were asked to respond on a scale from 1 (strongly disagree) - 7 (strongly agree). We show the results from this survey question in Fig. 2, again with standard error bars. Users rated PowerShell as more helpful in all three tasks, with the *Move Files* task showing the most significant difference ($p < .01$).

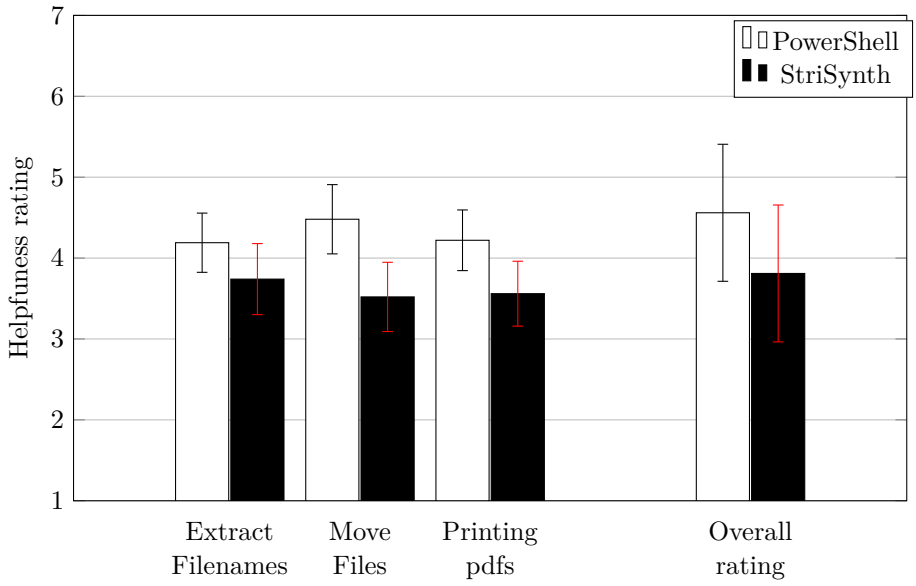

Figure 2 Users' (N=27) self reported measure of the helpfulness of each tool with standard error bars.

The results in Fig. 2 show the surprising insight that, despite the efficiency of StriSynth as demonstrated in Fig. 1, users perceived PowerShell to be the more helpful tool. Unfortunately, as we did not anticipate such unexpected results, our study design did not include a more detailed definition of helpfulness, or ask users to give a more detailed description of their interpretation of what it means for a tool to be helpful. We can however, at least surmise that efficiency is not a complete proxy measure for helpfulness.

4.3 Impact of prior user experience

Our study asked users to self-report their prior experience with scripting languages in a post-study survey. The survey used a seven-point Likert scale for users assess the experience. Fig. 3 shows the distribution of experience in three categories for all users.

To understand the impact of prior experience on how the users interacted with StriSynth, we split our user population into two categories. We have the inexperienced user group, which is the users who rated their prior experience with PowerShell as a 1 (unfamiliar), and the complement set of users as the experienced user group, who rated their prior experience with PowerShell as (≥ 2). In Fig. 4, we show how these two groups performed in the study.

Fig. 4 shows that both groups of users completed the tasks faster with StriSynth. A more subtle and interesting insight is that inexperienced users had a greater relative speedup in task completion when using StriSynth. That is, inexperienced users benefited more from using StriSynth as compared to the benefit to experienced users. This provides evidence for the widely stated perception that programming by example is a domain well-suited for novice programmers.

4.4 Threats to Validity

In a usability study, it is important to avoid any possible selection bias in the call for participants. Selection bias can be an issue if the set of users selected systematically differs

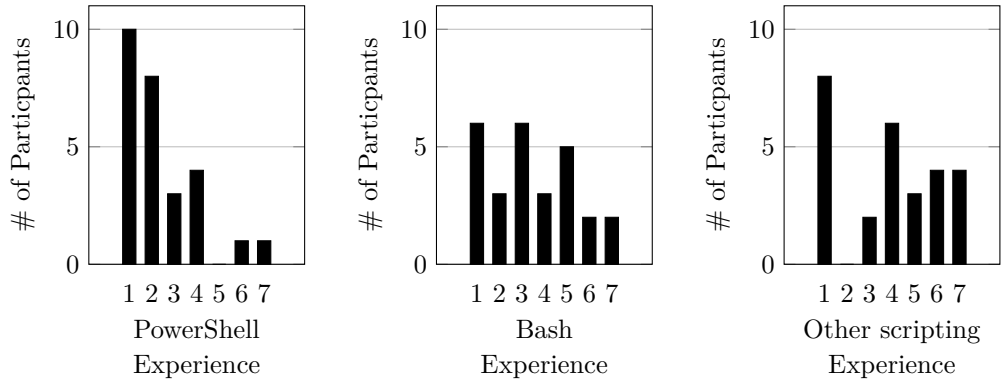

Figure 3 Users' (N=27) self reported prior experience with various scripting languages from 1 (Unfamiliar) to 7 (Expert User).

from the target population. The results we have presented are from a set of users that work as professional IT support specialists. We do not believe that we have any selection bias here because in this work, we specifically wanted to explore the impact synthesis can have in the real-world on such professionals.

A further potential threat to the validity of our results is in the social desirability bias, or need-to-please phenomena, whereby users will subconsciously try to produce the results they expect the researcher would like to see. This potential bias can occur when users are asked to compare a tool that is a known standard with an alternative that the user knows to be developed by the researcher. To do combat this issue, we presented StriSynth and PowerShell as tools named StriSynthA and StriSynthB. In this way, we framed the study as a comparison between two different tools that we had developed, eliminating the potential need-to-please bias. This was a critical component to our study design that allowed us to observe the disconnect between efficiency and users' perceived helpfulness of each tool.

5 Application to Related Work

The results from our user study are specifically targeted at the impact of programming by example systems for scripting in IT professional populations. We discuss here how our results can be interpreted and extended to other PBE domains.

Gulwani et al. [12] show that PBE is an effective paradigm for industrial application in spreadsheet manipulation, such as string transformations [1, 10, 9], table transformations [11] and database look-ups [28]. Another approach is based on the abstraction of 'topes' [27], which lets users create abstractions for different data present in a spreadsheet. With topes, a programmer uses a GUI to define constraints on the data, and to generate a context-free grammar that is used to validate and reformat the data. These application domains of PBE are focused on a similar population of non-expert programmers, and so it may be possible to observe a similar efficiency vs helpfulness phenomena.

Unlike programming by example, in which the user provides input-output examples, programming by demonstration is characterized by the user providing a complete trace demonstration leading from the initial to the final state. There are several programming by demonstration systems [6], such as Simultaneous Editing [25] for string manipulation, SMARTedit [22] for text manipulation and Wrangler [17] for table transformations. As programming by demonstration requires intermediate configurations instead of just input and

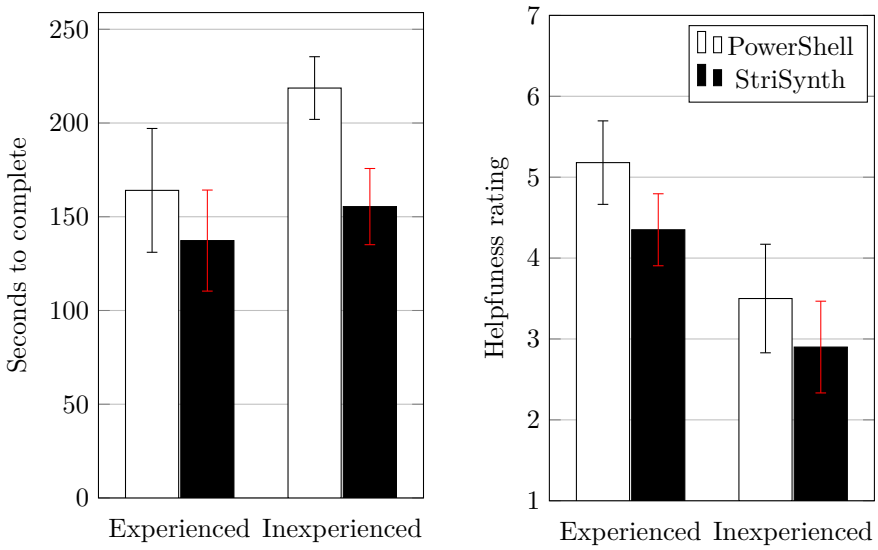

Figure 4 We grouped users as Experienced (PowerShell experience≥2, N=17) and Inexperienced (PowerShell experience=1, N=10). We report average time to complete the tasks, and self reported helpfulness of the tools, as separated by these two groups.

output examples, this paradigm is usually less flexible [21] than programming by example, but the synthesis problem is easier. Based on our results here, it is possible that this reduced flexibility may indicate users would rate programming by demonstration even less helpful (but possible more efficient) than PBE in certain domains.

The Myth [26] and Λ^2 [7] systems support PBE for inductively defined data-types in functional languages. In contrast to StriSynth which focuses on scripting tasks, these tools are focused on synthesis for more general purpose programming languages. The results from our study may be cautiously extrapolated other domains - while the theme of PBE is the same, interaction preference for users may differ when looking at general purpose languages.

Instead of providing specification in terms of examples or demonstrations, specification can also be given in more formal and complete ways. InSynth [15, 14], CodeHint [8] and the C# code snippets on demand [31] are systems that aim to provide code snippets based on context such as the inferred type or the surrounding comments. Leon [20] and Comfusy [19, 18] synthesize code snippets based on complete specifications, which are written in the same language that is used for programming. Sketch [29] takes as input an incomplete program with holes, and synthesizes code to complete the so that it meets the specification. These techniques provide a more nuanced interface that may seem, from a perspective of helpfulness, to be more similar to a traditional language paradigm.

6 Conclusions

Our study shows that users do not always correlate an efficient programming language with a helpful language. A more thorough exploration of this finding requires a follow up study, in particular to discover the definition of helpfulness that participants are using. A key question to answer would be whether users had erroneously perceived PowerShell to be more efficient and therefore helpful, or if users consciously have other metrics in mind that constitute the helpfulness of language. One intuitive interpretation may be tied to the interface style of the

paradigm - StriSynth and other PBE tools are generally limited in their ability to directly work with a traditional programming language and use familiar concepts such as variables and loops. This may seem to make for a less helpful language for new users.

References

1 Flash Fill (Microsoft Excel 2013 feature). URL: http://research.microsoft.com/users/sumitg/flashfill.html.

2 Stack Overflow: Auto increment a variable in regex. http://goo.gl/GPuZP3. Accessed: 2015-03-25.

3 Stack Overflow: What is the most difficult/challenging regular expression you have ever written? http://goo.gl/LLJe0r. Accessed: 2015-03-24.

4 Super User: How to batch combine jpeg's from folders into pdf's? http://goo.gl/LnGYH7. Accessed: 2015-05-13.

5 Sebastian Burckhardt, Manuel Fähndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! Continuous feedback in UI programming. In *PLDI*, 2013. doi:10.1145/2491956.2462170.

6 A. Cypher and D.C. Halbert. *Watch what I Do: Programming by Demonstration*. MIT Press, 1993.

7 John Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing Data Structure Transformations from Input-Output Examples. In *PLDI*, 2015.

8 Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. CodeHint: dynamic and interactive synthesis of code snippets. In *ICSE*, 2014. doi:10.1145/2568225.2568250.

9 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011. doi:10.1145/1926385.1926423.

10 Sumit Gulwani. Synthesis from Examples. *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*, 10(2), 2012. Invited talk paper.

11 Sumit Gulwani. Synthesis from Examples: Interaction Models and Algorithms. In *SYNASC*, 2012. doi:10.1109/SYNASC.2012.69.

12 Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8), 2012. doi:10.1145/2240236.2240260.

13 Sumit Gulwani, Mikael Mayer, Filip Niksic, and Ruzica Piskac. StriSynth: Synthesis for Live Programming. In *ICSE*, 2015.

14 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, 2013. doi:10.1145/2491956.2462192.

15 Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive Synthesis of Code Snippets. In *CAV*, 2011. doi:10.1007/978-3-642-22110-1_33.

16 Reiner Hähnle and Marieke Huisman. 24 Challenges in Deductive Software Verification. In Giles Reger and Dmitriy Traytel, editors, *ARCADE 2017. 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements*, volume 51 of *EPiC Series in Computing*. EasyChair, 2017. doi:10.29007/j2cm.

17 Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, 2011. doi:10.1145/1978942.1979444.

18 Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Comfusy: A Tool for Complete Functional Synthesis. In *CAV*, 2010. doi:10.1007/978-3-642-14295-6_38.

19 Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, 2010. doi:10.1145/1806596.1806632.

20 Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Software synthesis procedures. *Commun. ACM*, 55(2), 2012. doi:10.1145/2076450.2076472.

21 Tessa Lau. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine*, 30(4), 2009. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2262.

22 Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. In *K-CAP*, 2003. doi:10.1145/945645.945654.

23 H. Lieberman. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

24 Karl Mazurak and Steve Zdancewic. ABash: Finding bugs in bash scripts. In *In ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2007.

25 Robert C. Miller and Brad A. Myers. Interactive Simultaneous Editing of Multiple Text Regions. In *USENIX*, 2001. URL: http://www.usenix.org/publications/library/proceedings/usenix01/miller.html.

26 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

27 Christopher Scaffidi, Brad A. Myers, and Mary Shaw. Topes: reusable abstractions for validating data. In *ICSE*, 2008. doi:10.1145/1368088.1368090.

28 Rishabh Singh and Sumit Gulwani. Learning Semantic String Transformations from Examples. *PVLDB*, 5(8), 2012. doi:10.14778/2212351.2212356.

29 Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.

30 Ronald E Walpole, Raymond H Myers, Sharon L Myers, and Keying Ye. *Probability and statistics for engineers and scientists*, volume 5. Macmillan New York, 1993.

31 Yi Wei, Youssef Hamadi, Sumit Gulwani, and Mukund Raghothaman. C# code snippets on-demand, 2014. URL: http://codesnippet.research.microsoft.com.

QDB: From Quantum Algorithms Towards Correct Quantum Programs

Yipeng Huang[1]
Princeton University, USA
yipeng@cs.princeton.edu
https://orcid.org/0000-0003-3171-6901

Margaret Martonosi
Princeton University, USA
mrm@princeton.edu

Abstract

With the advent of small-scale prototype quantum computers, researchers can now code and run quantum algorithms that were previously proposed but not fully implemented. In support of this growing interest in quantum computing experimentation, programmers need new tools and techniques to write and debug QC code. In this work, we implement a range of QC algorithms and programs in order to discover what types of bugs occur and what defenses against those bugs are possible in QC programs. We conduct our study by running small-sized QC programs in QC simulators in order to replicate published results in QC implementations. Where possible, we cross-validate results from programs written in different QC languages for the same problems and inputs. Drawing on this experience, we provide a taxonomy for QC bugs, and we propose QC language features that would aid in writing correct code.

2012 ACM Subject Classification Computer systems organization → Quantum computing

Keywords and phrases correctness, debugging

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2018.4

1 Introduction

Quantum computing is reaching an inflection point. After years of work on both QC algorithms and low-level QC devices, small but viable QC prototypes are now available to run programs. These QC prototypes are increasing in size, with much research attention being placed on improving their reliability and increasing the counts of qubits (quantum bits), the fundamental building block for QC [11, 19, 31].

With small-scale machines available to run real code, a natural challenge lies in creating correct and useful programs to run on them [3, 12]. Until recently, QC algorithms were rarely programmed for actual execution, and therefore relatively little QC debugging has ever occurred. Furthermore, QC debugging faces challenges beyond that of classical computing. In particular, typical debugging approaches based on printing out variable values during program execution do not easily apply to QC programs, because program states in QC "collapse" to classical values when observed. Second, QC's "no cloning rule" precludes us from making a spare copy of variables to observe them elsewhere. Third, while we have more freedom to observe states in QC simulations on classical computers, the massive state spaces of QC executions limits this approach to small programs. Finally, even when limited simulations are tractable, it can be difficult to interpret the simulation results.

[1] This work is funded in part by EPiQC, an NSF Expedition in Computing, under grant 1730082

This paper surveys a range of QC algorithms and programs and offers a set of empirical and experiential insights on today's state-of-the-art in QC debugging. For three benchmarks representing different application areas, we perform detailed debugging based on small-scale simulations. For each, we give case studies of the types of bugs we found. Most importantly, we use these experiences to assemble a set of "design patterns for QC programming" and related best practices in QC debugging.

In particular, the contributions of this paper are as follows:

- We specifically explore three major areas: quantum chemistry, integer factorization, and database search. This is a broad spectrum of QC algorithms across not just application domains, but also problem size and algorithm strategies. This allows us to point out particular domain-specific challenges or opportunities.
- Where available, we study the same algorithm implemented in different languages or infrastructures. From this, we draw comparative insights regarding how programming language or environment support can be useful in QC programming and debugging.
- From these insights and experiences, we lay out a plan for debugging support in QC programming environments to aid users in creating quantum code. These include assertions, unit testing, code reuse, polymorphism, and QC-specific language types and syntax.

Overall, while QC programming has received significant prior attention and QC debugging has received some as well, our work offers steps forward in its detailed and comparative assessment across problem types and languages. We see our work offering useful insights for QC programmers themselves, as well as language and system designers interested in building next-generation compilers and debuggers.

2 Background on QC programming

First, we review the principles of quantum computing [14, 22, 23, 26], in order to understand how writing correct quantum programs is different from classical programming.

2.1 Qubits, superpositions, and entanglement

The basic unit of information in QC is the qubit, which can take on values of $|0\rangle$ and $|1\rangle$ like bits in classical computing, but can also be viewed as a probabilistic "superposition" between the two values. Quantum computers can also "measure" the value of a qubit, forcing it to collapse out of superposition into a classical value such as '0' or '1'. Measurement disturbs the values of variables in a quantum computer, so we cannot easily pause execution and observe the values of qubits as a quantum program runs.

The state of individual qubits can be "entangled" together. For this reason, as more qubits come into play in a quantum computer, the number of states that data can be in grows exponentially. For example, a two-qubit system can take on the values $|00\rangle, |01\rangle, |10\rangle, |11\rangle$, along with superpositions among these values; furthermore, the two qubits can even be in a state of entanglement where the two cannot be treated as independent pieces of information. A three qubit system has potential superpositions of eight states, and so on. This exponential growth of possible values underlies the power of QC.

As a result of this large number of possible states, running a quantum program in simulation on a classical computer is costly. Naive simulation of a 20-qubit quantum computer, for example, needs 2^{20} or roughly one million floating point numbers just to store the program state at any instant. For this reason, testing and debugging quantum programs in simulation is only possible for toy-sized programs.

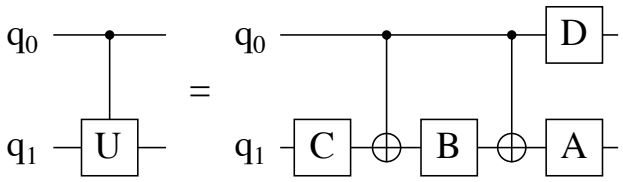

Figure 1 Decomposition of a simple QC program. Time flows left to right, showing sequences of operations applied to qubits q_0 and q_1. The left program is a "controlled" arbitrary operation U, which means whether the operation U works on q_1 is dependent on the value of q_0. The left sequence decomposes into the equivalent right sequence of more basic operations. The basic operations include single-qubit "rotations" A through D that alter the probability distribution of qubit values. The operations also include two two-qubit "CNOT" operations that flip a qubit (denoted $\oplus$) contingent on the value of another qubit (denoted $\bullet$) [26].

2.2 Quantum computer operations, programs, and a taxonomy for bugs

The process of quantum computing involves applying operations on qubits. We use diagrams such as Figure 1 to represent sequences of quantum operations. Looking at Figure 1 we see that quantum programs consist of three conceptual parts [8]:

1. **Inputs** to quantum algorithms include *classical* input parameters such as coefficients for rotations A through D, and *quantum* initial values for qubits such as q_0 and q_1.
2. **Operations**, such as the specification of how a complex operation such as controlled arbitrary operation U (Figure 1, left) decomposes into basic operations A through D and CNOTs (Figure 1, right). Additionally, both basic and complex operations can be further composed according to patterns such as iteration, recursion, and mirroring.
3. **Outputs** of quantum algorithms are the final classical measurement values of qubits such as q_0 and q_1. Furthermore, any temporary variables used in the course of a program have to be safely disentangled from the rest of the quantum state and discarded.

Bugs in quantum programs can crop up due to mistakes made in any of these three parts of a QC program. We will give examples of each kind of bug along with how to prevent them, using detailed case studies in the rest of this paper.

2.3 QC algorithm primitives, benchmarks, and open source frameworks

Given the rapid growth of QC infrastructure, we now have a chance to test a variety of quantum algorithms written in many languages [18]. Many different quantum algorithms rely on a handful of QC algorithm primitives to get speedups relative to classical algorithms [4, 24, 25]. Table 1 classifies canonical quantum algorithms according to their algorithm primitives, and cites example implementations in different QC languages and tool chains.

This paper specifically focuses on program bugs and defenses in three areas: a quantum chemistry problem that uses quantum phase estimation, integer factorization using Shor's order finding algorithm, and Grover's database search algorithm.

Using programs written in the Scaffold language as a starting point [13], we compile Scaffold code to OpenQASM, a QC assembly language [5]. Then, we simulate the programs operation-by-operation in the QX simulator [15], in order to see their intermediate states and outputs. We cross reference the programs' results against implementations in other languages, such as LIQUi|> [32], ProjectQ [10, 36] and Q# [37]. From this debugging experience we identify possible bugs and defenses. Furthermore, we review the codes across languages to understand the relative merits of different QC language features.

Table 1 Quantum algorithm primitives and open source benchmarks in open source tool chains.

Primitives	Quantum algorithms	Benchmark implementations
Entanglement protocols	superdense coding / quantum teleportation	Q# teleportation [37] pyQuil teleportation [35]
Quantum (random) walks	tree traversal	Scaffold / Quipper binary welded tree [6, 13, 39]
	graph traversal	Scaffold / Quipper triangle finding problem [6, 13, 39]
	satisfiability	Scaffold / Quipper Boolean formula [6, 13, 39]
Adiabatic	Ising spin model	Scaffold / Q# adiabatic Ising model [13, 37]
	quantum approximate optimization algorithm	QISKit Aqua QAOA pyQuil QAOA ansatz [35]
Variational Quantum Eigensolver	Hamiltonian simulation	QISKit Aqua quantum chemistry Q# H_2 simulation [37] Rigetti Grove VQE [35]
Quantum Fourier Transform (QFT)	phase estimation	Scaffold / Quipper ground state estimation [6, 13, 39]
	period finding	Scaffold class number [13]
	order finding	Scaffold / ProjectQ / Q# Shor's factoring [13, 36, 37]
	hidden subgroup problem	Quipper unique shortest vector [6, 39]
	linear algebra	Quipper quantum linear systems [6, 39]
Amplitude amplification	database search	Scaffold square root [13] ProjectQ / Q# Grover's database search [36, 37]

3 Case study: Quantum chemistry

First, we discuss our experience building up and debugging a simple quantum chemistry program. Quantum chemistry problems entail finding properties of molecules from theoretical first principles [20, 27]. Researchers anticipate these will be the first applications for QC due to the relatively few number of qubits they need to surpass classical computer algorithms. Debugging these problems is distinctively challenging, due to the importance of getting a large number of classical input parameters all correct, and because of the dearth of physically meaningful intermediate states we can check in the course of algorithm execution.

3.1 Bug type 1: Incorrect classical input parameters

A key part of quantum chemistry programs is in correctly building up a "Hamiltonian" subroutine that simulates inter-electron forces. The procedure for doing this was laid out in detail by Whitfield [41]. We followed this procedure to create a subroutine for simulating the hydrogen molecule, but we needed additional validation from several other sources to get a bug-free subroutine [40]. These resources include raw chemistry data found in open source repositories for the LIQUi|> framework[2]. The final parameters for actual operations on qubits were validated against a follow-up paper [33] and an implementation in the QISKit framework[3]. Because the procedure for preparing these quantum chemistry models involves many steps and needs domain expertise, software packages such as OpenFermion now automate this process [21]. Nonetheless, there is room for improvement in standardizing input data formats to eliminate bugs in this process.

[2] `https://github.com/StationQ/Liquid/blob/master/Samples/h2_sto3g_4.dat`
[3] `https://github.com/Qiskit/aqua/blob/master/test/H2-0.735.json`

Table 2 QC calculated energy for H_2 (bond length = 73.48 pm) for different electron assignments.

	Electron assignments				**QC calculated energy (relative)**
	Bonding		Antibonding		
	↑	↓	↑	↓	
3rd excited state (E3)	0	0	1	1	-0.164
2nd excited state (E2)	0	1	1	0	-0.217
	1	0	0	1	
1st excited state (E1)	0	1	0	1	-0.244
	1	0	1	0	
Ground state (G)	1	1	0	0	-0.295

Once the Hamiltonian subroutine is built, we can use the model in a variety of quantum algorithms spanning different primitives in Table 1. These include phase estimation (an application of quantum Fourier transforms) [28], variational quantum eigensolvers [30], and adiabatic algorithms [1]. In this paper, we use iterative phase estimation to find the ground state energy of our H_2 model, validating results published by Lanyon [17].

3.2 Bug type 2: Incorrect quantum initial values

The correct preparation of qubit initial values is important. Incorrect initial values would cause the program to find solutions to different problems altogether. In this quantum chemistry problem, the initial values control the locations of the two electrons in H_2. As shown in Table 2, we need the qubit assignment for finding the ground energy of H_2, while other assignments lead to results for other energy levels.

The symmetry of H_2 allows us to perform a sanity check, to make sure the Hamiltonian and the iterative phase estimation subroutines are working correctly. Though there are six ways to assign two electrons to four locations, there are in fact only four distinct energy levels, as shown in the experimental data. Checking that the two different ways to obtain E1 (and E2) give the same energy levels validates that the model correctly preserves symmetry.

3.3 Defense type 1: Assertions on algorithm preconditions

Given how important correct initial values are for all quantum algorithms, it is worthwhile to explicitly check for these algorithm preconditions before continuing with execution or simulation. What the preconditions should be depends on the type of algorithm. For example, the phase estimation subroutine in this case study (along with other algorithms relying on quantum Fourier transforms), expect inputs that are maximally in superposition among all possible values. Likewise, "ancillary qubits" such as the inputs to the Hamiltonian subroutine take on completely classical (integer) initial values. Lastly, quantum protocols often need to start with entangled states. These required input states are among the few places in quantum algorithms where we can check states for specific values. We can check these preconditions by running or simulating programs up to the entry point of subroutines, and performing a premature measurement to check for these anticipated states, finally restarting the program knowing that execution is correct up to that point. Thus far, the Q# framework has the most extensive support for precondition checking [37].

Table 3 Shor's factorization algorithm subroutines [23, p. 25].

Program subroutine code	Shared library code
Shor's routine for factoring 15; calculating powers of a number - controlled modular multiplication - controlled modular addition - controlled addition	- quantum Fourier transform - controlled controlled rotation - controlled rotation - controlled swap - swap

3.4 Defense type 2: Assertions on algorithm progress

Unlike the other two case studies later in this paper, the debugging process for the quantum chemistry benchmark is coarse-grained. That is because the Hamiltonian subroutine is a monolithic block of code whose components do not have obvious expected outputs—its components represent pair-wise electron interactions, and do not have inherent physical meaning. So how do we debug this program? The preconditions in the last section make sure the inputs to the algorithm are correct; the other observable state we have for debugging is to check the behavior of the algorithm as a whole.

In this quantum chemistry program, we can check for two types of overall algorithm behavior. One is the solution should converge to a steady value as finer Trotter time steps (a kind of numerical approximation) are chosen; a lack of this type of convergence indicates a bug in the Hamiltonian subroutine. The other algorithm behavior is when we vary the precision of the phase estimation algorithm, the most significant bits of the measurement output sequences should be the same—in other words, rounding the output of a high-precision experiment should yield the same output as a lower-precision experiment. a lack of this convergence indicates a bug in the iterative phase estimation subroutine. These checks for expected algorithm progress also apply to other algorithms.

4 Case study: Shor's algorithm for integer factorization

While our debugging strategy for quantum chemistry had to be coarse-grained, the debugging process for Shor's algorithm in this section allows us to look inside the program one subroutine at a time, where we can compare the intermediate results against known expected values.

Shor's factorization algorithm uses a quantum computer to factor a composite number in polynomial time complexity, providing exponential speedup relative to the best known classical algorithms [34]. We follow an example for an implementation that minimizes the qubit cost [2], and replicate results for factoring 15, the simplest example [16] [26, p. 235].

4.1 Bug type 3: Incorrect operations and transformations

In order to correctly implement Shor's algorithm we first have to build up the quantum subroutines shown in Table 3. These basic subroutines can be tricky to get right. Take the controlled rotation in Figure 1 as an example: Table 4 shows multiple ways to code the decomposition of the controlled rotation, and small mistakes can lead to incorrect behavior.

Table 4 Correct and incorrect code for rotation decomposition. Using the Scaffold language [13] as an example, we code out the controlled operation U in Figure 1 where U is a rotation in just one axis. Because only one axis is needed, we can drop either operation A or C, paying attention to the sign on the angles. Reordering the lines of code or signs results in a rotation in the wrong direction.

Correct, operation A unneeded	Correct, operation C unneeded	Incorrect, angles flipped
`Rz(q1,+angle/2); // C`	`CNOT(q0,q1);`	`Rz(q1,-angle/2);`
`CNOT(q0,q1);`	`Rz(q1,-angle/2); // B`	`CNOT(q0,q1);`
`Rz(q1,-angle/2); // B`	`CNOT(q0,q1);`	`Rz(q1,+angle/2);`
`CNOT(q0,q1);`	`Rz(q1,+angle/2); // A`	`CNOT(q0,q1);`
`Rz(q0,+angle/2); // D`	`Rz(q0,+angle/2); // D`	`Rz(q0,+angle/2); // D`

Listing 1 Controlled adder subroutine using Fourier transform in the Scaffold language [13].

```
// outputs a + b, where a is a 'width' bit constant integer
// b is an integer encoded on 'width' qubits in Fourier space
module cADD (
  const unsigned int c_width, // number of control qubits
  qbit ctrl0, qbit ctrl1, // control qubits
  const unsigned int width, const unsigned int a, qbit b[]
) {
  for (int b_indx=width-1; b_indx>=0; b_indx--) {
    for (int a_indx=b_indx; a_indx>=0; a_indx--) {
      if ((a >> a_indx) & 1) { // shift out bits in constant a
        double angle = M_PI/pow(2,b_indx-a_indx); // rotation angle
        switch (c_width) {
          case 0: Rz ( b[b_indx], angle ); break;
          case 1: cRz ( ctrl0, b[b_indx], angle ); break;
          case 2: ccRz ( ctrl0, ctrl1, b[b_indx], angle ); break;
}}}}}
```

4.2 Defense type 3: Language support for subroutines / unit tests

An obvious defense against coding mistakes in basic subroutines is to use a library of shared code. Doing so helps ensure program correctness by allowing programmers to exhaustively validate small subroutines, in order to bootstrap larger subroutines. Unit testing is especially important in QC as running or simulating large quantum programs is impossible for now.

An additional benefit is logically structured code allows compilers to select the best concrete implementation for the abstract functionality the programmer needs, based on hardware constraints and input parameters [8]. For example, the most cost-efficient implementation for modular exponentiation in Shor's factorization algorithm depends on how many qubits are available: the compiler can choose from minimum-qubit [2, 9, 38] or minimum-operation [29] implementations for the arithmetic subroutines.

4.3 Bug type 4: Incorrect composition of operations using iteration

Once we have built our basic subroutines, a common pattern in quantum programs is to use iterations to compose subroutines. Listing 1 shows the iteration code for a constant-value adder, showing tricky places in lines 8 through 11 for bugs to crop up, including indexing errors, bit shifting errors, endian confusion, and mistakes in rotation angles. In general this type of iteration code is commonplace in programs that rely on quantum Fourier transforms.

Table 5 Correct classical input a and a^{-1} to Shor's algorithm for factoring 15, using 7 as a guess.

k, the algorithm iteration	0	1	2	3	...
$a = 7^{2^k} \mod 15$	7	4	1	1	...
a^{-1}; $a \times a^{-1} \equiv 1 \mod 15$	13	4	1	1	...

4.4 Defense type 4: Language support for numerical data types

One way to defend against bugs in iteration code is to introduce QC data types for numbers, providing greater abstraction than working with raw qubits. For example, ProjectQ has quantum integer data types [36], while Q# [37] and Quipper [6, 39] offer both big endian and little endian versions of subroutines involving iterations. These QC data types permit useful operators (e.g., checking for equality) that help with debugging and writing assertions.

4.5 Bug type 5: Incorrect deallocation of qubits

Variable scoping is an important language feature in classical computing that ensures proper data encapsulation. In QC, scoping is similarly important for temporary variables known as "ancillary qubits." Anything that happens to a subroutine's ancillary qubits—such as measurement, reinitialization, or lapsing into decoherence—may have unintended effects on the subroutine's outputs[4]. Because improper ancillary qubit deallocation can lead to wrong results, it is important for subroutines to reverse their operations on their ancillary qubits, so that they properly undo any entanglement between the ancillary and output qubits.

We can demonstrate a bug involving incorrect qubit deallocation, by deliberately making a mistake while reversing operations in a subroutine. For example, Shor's algorithm relies on correct pairs modular inverse numbers as input parameters, such as those in Table 5. By feeding an incorrect pair of inputs (e.g., replacing 13 with a 12), the algorithm proceeds to possibly give us wrong output values, as shown in Table 6. At the same time, the mistake prevents the modular multiplication operation from being properly reversed, which has the effect of preventing the ancillary qubits from properly disentangling with other qubits, so they fail to return to their initial values at the end of the algorithm.

4.6 Defense type 5: Assertions on algorithm postconditions

We can use postconditions at the end of algorithms to detect bugs that lead to incorrect deallocation of ancillary qubits. Continuing with our example in Table 6, we see that the cases where ancillary qubits collapse to anything other than zero correspond to cases where the outputs are wrong. That is because the ancillary qubits remain improperly entangled with the output qubits at the end of the algorithm. We can detect these buggy outputs by asserting that ancillary qubits should always return to their initial values. The significance of these observations is that when algorithms work correctly, we typically do not care to measure the value of ancillary qubits as they do not contain information. But in buggy QC algorithm implementations, they are useful side channels for debugging.

[4] As an analogy in classical computing, it is as if accessing an out-of-scope variable can still affect program state; while such behavior is unintuitive, it is a result of how entanglement works in QC.

Table 6 Probability of measuring values of outputs and ancillary qubits of Shor's algorithm, with incorrect inputs ($a^{-1} = 12$ instead of 13 on first iteration). If the ancillary qubits collapse to zero on measurement, the algorithm still succeeds, returning correct outputs of 0, 2, 4, 6 [26, p. 235]. However, the possibility of measuring non-zero for the ancillary qubits indicates a bug.

Probability		Output measurement							
		0	1	2	3	4	5	6	7
Ancillary qubit measurement	0	1/8	0	1/8	0	1/8	0	1/8	0
	2	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	7	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	8	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64
	13	1/64	1/64	1/64	1/64	1/64	1/64	1/64	1/64

Table 7 Grover's amplitude amplification subroutine in two languages, showcasing QC-specific language syntax for reversible computation (rows 2 & 6) and controlled operations (rows 3 & 5).

	Scaffold (C syntax) [13]	ProjectQ (Python syntax) [36]

1	`int j;`
	`qbit ancilla[n-1]; // scratch register`
	`for(j=0; j<n-1; j++) PrepZ(ancilla[j],0);`
		`# reflection across`
		`# uniform superposition`

2	`// Hadamard on q`
	`for(j=0; j<n; j++) H(q[j]);`
	`// Phase flip on q = 0...0 so invert q`
	`for(j=0; j<n; j++) X(q[j]);`
		`with Compute(eng):`
		`  All(H) | q`
		`  All(X) | q`

3	`// Compute x[n-2] = q[0] and ... and q[n-1]`
	`CCNOT(q[1], q[0], ancilla[0]);`
	`for(j=1; j<n-1; j++)`
	`  CCNOT(ancilla[j-1], q[j+1], ancilla[j]);`
		`with Control(eng, q[0:-1]):`

4	`// Phase flip Z if q=00...0` `cZ(ancilla[n-2], q[n-1]);`	`  Z | q[-1]`

5	`// Undo the local registers`
	`for(j=n-2; j>0; j-)`
	`  CCNOT(ancilla[j-1], q[j+1], ancilla[j]);`
	`CCNOT(q[1], q[0], ancilla[0]);`
		`# ProjectQ automatically`
		`# uncomputes control`

6	`// Restore q`
	`for(j=0; j<n; j++) X(q[j]);`
	`for(j=0; j<n; j++) H(q[j]);`
		`Uncompute(eng)`

5 Case study: Grover's algorithm for database search

So far, we have presented defenses against bugs following two general strategies. One is to use assertions to detect when and where the program has a bug. The other is to use quantum-specific programming language features to prevent bugs altogether: these features include support for subroutines and numerical types for quantum data. Here in this section, we use the Grover's benchmark to showcase two more language features for common QC program patterns: reversible computation and controlled operations.

Grover's search algorithm finds an entry that matches search criteria, among an input data set of size N, with a time cost on the order of $\sqrt{N}$. That represents a polynomial speedup relative to the linear time cost in a classical computer [7].

The Grover's algorithm comprises three parts. First, the input qubits representing the indices of the matching entries are put in a state of superposition, akin to querying all entries at once. Second, the queries are put through a subroutine that checks for the search criteria.

In this case study, our criteria is to find the square root of a number in a Galois field of two elements, a simple abstract algebra setting. Finally in the critical step, the amplitude amplification algorithm primitive amplifies the index that matches the criteria while damping out those that do not. The operations in this final step are prime examples of two QC program patterns, reversible computation and controlled operations. We show in Table 7 how these code patterns are written in two languages, Scaffold [13] and ProjectQ [36].

5.1 Bug type 6: Incorrect composition of operations using mirroring

Section 4.5 discussed how bugs in deallocating ancillary qubits can happen due to bad parameters. Here we see how bugs in deallocating ancillary qubits can happen due to incorrect composition of operations following a mirroring pattern. For example, in Table 7, the operations in rows 2 and 3 are respectively mirrored and undone in rows 6 and 5. These lines of code need careful reversal of every loop and every operation.

5.2 Defense type 6: Language support for reversible computation

Syntax support for reversible computation, such as that in ProjectQ [36], automatically mirrors and inverts sequences of operations, shortening code and reducing mistakes.

5.3 Bug type 7: Incorrect composition of operations using recursion

A common pattern in quantum programs involves performing operations (e.g., add), contingent on a set of qubits known as control qubits. Without language support, this pattern needs many lines of code and manual allocation of ancillary qubits. In the Scaffold code example in Table 7, rows 3 and 5 are just computing the intersection of qubits `q`, with the help of ancillary qubits initialized in row 1, in order to realize the controlled rotation operation in row 4. Furthermore, quantum algorithms often need varying numbers of control qubits in different parts of the algorithm, leading to replicated code from multiple versions of the same subroutine differing only by the number of control qubits[5].

5.4 Defense type 7: Language support for controlled operations

Language support for controlled operations (e.g, ProjectQ) shortens code, preventing mistakes.

6 Conclusion

For the first time, we have access to comprehensive and representative program benchmarks for all major areas of quantum algorithms, implemented in multiple languages, along with input datasets and outputs that are detailed enough to permit cross-validation. Using our experience running and debugging these programs, we presented in this paper defense strategies that facilitate writing bug-free QC code, summarized in Table 8. Successful transplantation of these ideas from classical languages to QC languages can pave the way towards correct and useful quantum programs.

[5] An example appeared in the Shor's case study Listing 1. The addition operation was contingent on control qubits taken as parameters in lines 4 and 5. Depending on how many control qubits were needed, the switch statement in lines 12 through 15 applied the correct operation.

Table 8 Applicability of defense strategies (down) against location of QC program bugs (across).

		input		operations				output
		classical params. §3.1	qubit alloc. §3.2	basic §4.1	iterate §4.3	mirror §5.1	recurse §5.3	qubit dealloc. §4.5
QC specific lang. features	unit testing §4.2		✓	✓	✓	✓	✓	✓
	data types §4.4				✓			
	reverse comp. §5.2				✓	✓		✓
	controlled ops. §5.4				✓	✓	✓	
Assertion checks	preconditions §3.3		✓					
	algo progress §3.4	✓	✓	✓	✓	✓	✓	✓
	postconds. §4.6	✓	✓	✓	✓	✓	✓	✓

References

1 R. Barends, A. Shabani, L. Lamata, J. Kelly, A. Mezzacapo, U. Las Heras, R. Babbush, A. G. Fowler, B. Campbell, Yu Chen, Z. Chen, B. Chiaro, A. Dunsworth, E. Jeffrey, E. Lucero, A. Megrant, J. Y. Mutus, M. Neeley, C. Neill, P. J. J. O'Malley, C. Quintana, P. Roushan, D. Sank, A. Vainsencher, J. Wenner, T. C. White, E. Solano, H. Neven, and John M. Martinis. Digitized adiabatic quantum computing with a superconducting circuit. *Nature*, 534:222 EP–, June 2016. doi:10.1038/nature17658.

2 Stephane Beauregard. Circuit for Shor's Algorithm Using 2N+3 Qubits. *Quantum Info. Comput.*, 3(2):175–185, March 2003. URL: http://dl.acm.org/citation.cfm?id=2011517.2011525.

3 Frederic T. Chong, Diana Franklin, and Margaret Martonosi. Programming languages and compiler design for realistic quantum hardware. *Nature*, 549:180 EP–, September 2017. doi:10.1038/nature23459.

4 Patrick J. Coles, Stephan Eidenbenz, Scott Pakin, Adetokunbo Adedoyin, John Ambrosiano, Petr M. Anisimov, William Casper, Gopinath Chennupati, Carleton Coffrin, Hristo Djidjev, David Gunter, Satish Karra, Nathan Lemons, Shizeng Lin, Andrey Y. Lokhov, Alexander Malyzhenkov, David Mascarenas, Susan M. Mniszewski, Balu Nadiga, Dan O'Malley, Diane Oyen, Lakshman Prasad, Randy Roberts, Philip Romero, Nandakishore Santhi, Nikolai Sinitsyn, Pieter Swart, Marc Vuffray, Jim Wendelberger, Boram Yoon, Richard J. Zamora, and Wei Zhu. Quantum Algorithm Implementations for Beginners. *CoRR*, abs/1804.03719, 2018. arXiv:1804.03719.

5 A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. Open Quantum Assembly Language. *ArXiv e-prints*, July 2017. arXiv:1707.03429.

6 Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 333–342, New York, NY, USA, 2013. ACM. doi:10.1145/2491956.2462177.

7 Lov K Grover. From Schrödinger's equation to the quantum search algorithm. *Pramana*, 56(2-3):333–348, 2001.

8 T. Häner, T. Hoefler, and M. Troyer. Using Hoare logic for quantum circuit optimization. *ArXiv e-prints*, September 2018. arXiv:1810.00375.

9 Thomas Häner, Martin Roetteler, and Krysta M. Svore. Factoring Using 2N + 2 Qubits with Toffoli Based Modular Multiplication. *Quantum Info. Comput.*, 17(7-8):673–684, June 2017. URL: http://dl.acm.org/citation.cfm?id=3179553.3179560.

10 Thomas Häner, Damian S. Steiger, Mikhail Smelyanskiy, and Matthias Troyer. High Performance Emulation of Quantum Circuits. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 74:1–74:9, Piscataway, NJ, USA, 2016. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=3014904.3015003.

11 Aram Harrow. Why Now is the Right Time to Study Quantum Computing. *XRDS*, 18(3):32–37, March 2012. doi:10.1145/2090276.2090288.

12 Thomas Häner, Damian S Steiger, Krysta Svore, and Matthias Troyer. A software methodology for compiling quantum programs. *Quantum Science and Technology*, 3(2):020501, 2018. URL: http://stacks.iop.org/2058-9565/3/i=2/a=020501.

13 Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. ScaffCC: A framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 1:1–1:10, New York, NY, USA, 2014. ACM. doi:10.1145/2597917.2597939.

14 Phillip Kaye, Raymond Laflamme, and Michele Mosca. *An Introduction to Quantum Computing*. Oxford University Press, Inc., New York, NY, USA, 2007.

15 N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, and K. Bertels. QX: A high-performance quantum computer simulation platform. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '17, pages 464–469, 3001 Leuven, Belgium, Belgium, 2017. European Design and Automation Association. URL: http://dl.acm.org/citation.cfm?id=3130379.3130487.

16 B. P. Lanyon, T. J. Weinhold, N. K. Langford, M. Barbieri, D. F. V. James, A. Gilchrist, and A. G. White. Experimental Demonstration of a Compiled Version of Shor's Algorithm with Quantum Entanglement. *Phys. Rev. Lett.*, 99:250505, December 2007. doi:10.1103/PhysRevLett.99.250505.

17 B. P. Lanyon, J. D. Whitfield, G. G. Gillett, M. E. Goggin, M. P. Almeida, I. Kassal, J. D. Biamonte, M. Mohseni, B. J. Powell, M. Barbieri, A. Aspuru-Guzik, and A. G. White. Towards quantum chemistry on a quantum computer. *Nature Chemistry*, 2:106 EP–, January 2010. doi:10.1038/nchem.483.

18 R. LaRose. Overview and Comparison of Gate Level Quantum Software Platforms. *ArXiv e-prints*, July 2018. arXiv:1807.02500.

19 Norbert M. Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A. Landsman, Kenneth Wright, and Christopher Monroe. Experimental comparison of two quantum computing architectures. *Proceedings of the National Academy of Sciences*, 114(13):3305–3310, 2017. doi:10.1073/pnas.1618020114.

20 S. McArdle, S. Endo, A. Aspuru-Guzik, S. Benjamin, and X. Yuan. Quantum computational chemistry. *ArXiv e-prints*, August 2018. arXiv:1808.10402.

21 J. R. McClean, I. D. Kivlichan, K. J. Sung, D. S. Steiger, Y. Cao, C. Dai, E. Schuyler Fried, C. Gidney, B. Gimby, P. Gokhale, T. Häner, T. Hardikar, V. Havlíček, C. Huang, J. Izaac, Z. Jiang, X. Liu, M. Neeley, T. O'Brien, I. Ozfidan, M. D. Radin, J. Romero, N. Rubin, N. P. D. Sawaya, K. Setia, S. Sim, M. Steudtner, Q. Sun, W. Sun, F. Zhang, and R. Babbush. OpenFermion: The Electronic Structure Package for Quantum Computers. *ArXiv e-prints*, October 2017. arXiv:1710.07629.

22 N.D. Mermin. *Quantum Computer Science: An Introduction*. Cambridge University Press, 2007.

23 Tzvetan S. Metodi, Arvin I. Faruque, and Frederic T. Chong. Quantum Computing for Computer Architects, Second Edition. *Synthesis Lectures on Computer Architecture*, 6(1):1–203, 2011. doi:10.2200/S00331ED1V01Y201101CAC013.

24 Ashley Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2:15023, 2016.

25 Michele Mosca. Quantum algorithms. In *Encyclopedia of Complexity and Systems Science*, pages 7088–7118. Springer, 2009.

26 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 10th edition, 2011.

27 Jonathan Olson, Yudong Cao, Jonathan Romero, Peter Johnson, Pierre-Luc Dallaire-Demers, Nicolas Sawaya, Prineha Narang, Ian Kivlichan, Michael Wasielewski, and Alán Aspuru-Guzik. Quantum information and computation for chemistry. *arXiv preprint arXiv:1706.05413*, 2017.

28 S. Patil, A. JavadiAbhari, C. Chiang, J. Heckey, M. Martonosi, and F. T. Chong. Characterizing the performance effect of trials and rotations in applications that use Quantum Phase Estimation. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 181–190, October 2014. doi:10.1109/IISWC.2014.6983057.

29 Archimedes Pavlidis and Dimitris Gizopoulos. Fast Quantum Modular Exponentiation Architecture for Shor's Factoring Algorithm. *Quantum Info. Comput.*, 14:649–682, May 2014. URL: http://dl.acm.org/citation.cfm?id=2638682.2638690.

30 Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5:4213 EP–, July 2014. doi:10.1038/ncomms5213.

31 John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018. doi:10.22331/q-2018-08-06-79.

32 M. Roetteler, K. M. Svore, D. Wecker, and N. Wiebe. Design automation for quantum architectures. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1312–1317, March 2017. doi:10.23919/DATE.2017.7927196.

33 Jacob T. Seeley, Martin J. Richard, and Peter J. Love. The Bravyi-Kitaev transformation for quantum computation of electronic structure. *The Journal of Chemical Physics*, 137(22):224109, 2012. doi:10.1063/1.4768229.

34 Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997. doi:10.1137/S0097539795293172.

35 R. S. Smith, M. J. Curtis, and W. J. Zeng. A Practical Quantum Instruction Set Architecture. *ArXiv e-prints*, August 2016. arXiv:1608.03355.

36 Damian S. Steiger, Thomas Häner, and Matthias Troyer. ProjectQ: an open source software framework for quantum computing. *Quantum*, 2:49, January 2018. doi:10.22331/q-2018-01-31-49.

37 Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018, pages 7:1–7:10, New York, NY, USA, 2018. ACM. doi:10.1145/3183895.3183901.

38 Yasuhiro Takahashi and Noboru Kunihiro. A Quantum Circuit for Shor's Factoring Algorithm Using 2N + 2 Qubits. *Quantum Info. Comput.*, 6(2):184–192, March 2006. URL: http://dl.acm.org/citation.cfm?id=2011665.2011669.

39 Benoît Valiron, Neil J. Ross, Peter Selinger, D. Scott Alexander, and Jonathan M. Smith. Programming the Quantum Future. *Commun. ACM*, 58(8):52–61, July 2015. doi:10.1145/2699415.

40 Dave Wecker, Bela Bauer, Bryan K. Clark, Matthew B. Hastings, and Matthias Troyer. Gate-count estimates for performing quantum chemistry on small quantum computers. *Phys. Rev. A*, 90:022305, August 2014. doi:10.1103/PhysRevA.90.022305.

41 J. D. Whitfield, J. Biamonte, and A. Aspuru-Guzik. Simulation of electronic structure Hamiltonians using quantum computers. *Molecular Physics*, 109:735–750, March 2011. doi:10.1080/00268976.2011.552441.

Identifying Barriers to Adoption for Rust through Online Discourse

Anna Zeng
Stanford University, USA

Will Crichton
Stanford University, USA

Abstract

Rust is a low-level programming language known for its unique approach to memory-safe systems programming and for its steep learning curve. To understand what makes Rust difficult to adopt, we surveyed the top Reddit and Hacker News posts and comments about Rust; from these online discussions, we identified three hypotheses about Rust's barriers to adoption. We found that certain key features, idioms, and integration patterns were not easily accessible to new users.

2012 ACM Subject Classification Human-centered computing → Human computer interaction (HCI)

Keywords and phrases rust, programming language usability

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2018.5

1 Introduction

Rust is a new programming language designed to usher low-level programming into the modern era. Rust uses strong type systems and functional programming to execute programs efficiently while avoiding the many safety problems that plague C and C++. As an open-source project with support from Mozilla, the Rust ecosystem has grown rapidly over the last decade. Hundreds of companies deploy Rust in production, and thousands of developers regularly use Rust in their projects. However, Rust has a notoriously steep learning curve. A community survey in 2017 revealed that 25% of the people who tried Rust and dropped it felt the language was "too intimidating, too hard to learn, or too complicated" [13]. Prominent members of the Rust community have stated that Rust is *supposed* to be hard to learn: "Rust has never claimed that it is something you can learn in half a week" [6]. Even still, the learning curve is only one aspect of many (library support, tooling, compile times, etc. [9]) that influences the adoption of a programming language.

If Rust is hard to learn or use, the question becomes: how should its developers prioritize language features to drive adoption, addressing the key challenges facing its current or potential user base? A developer on the Rust compiler told us in an interview that these decisions are usually made in an ad hoc way, based on the intuitions of the language developers and occasional feedback from community members in a potpourri of online forums [14]. To address this issue, we sought to understand the challenges of adopting Rust by analyzing online discourse within the Rust community. We conducted a holistic survey of popular posts and comment threads about Rust to identify beliefs of developers using Rust that highlight ongoing issues in the practice of the language.

2 Methodology

Prior work on analyzing language adoption has focused on evaluating various languages and tools through questionnaires, surveys, in-person interviews, and automated code base analysis. For example, Meyerovich and Rabkin [9] combined all of the above to highlight both empirical popularity trends as well as reported beliefs of developers about a wide range of topics from types vs. tests to features vs. libraries. Christakis and Bird [2] used surveys and interviews to understand the adoption of program analyzers, and Ray et al. [10] used program analyzers to compare languages based on bugs detected in a large corpus of GitHub repositories.

This study differs from previous work in two ways: First, we focused only on a single language, Rust. While this focus potentially limits the generality of our insights, it allows us to produce deeper insights via thorough consideration of the factors driving specifically Rust's adoption. Second, we performed a content analysis on the existing discourse on Rust rather than creating a new survey. We observed that Rust is frequently written about on blogs, and subsequently discussed online by the people developing Rust itself (henceforth referred to as "Rust language developers"), by the people learning and using Rust ("Rust community"), and by the broader tech community. In fact, a version of this ethnomethodological approach [4] was already used by the Rust language developers to inform the Rust 2018 roadmap [12] and also used to analyze other online communities like Mechanical Turk [7].

We gathered a corpus of articles and corresponding comments from Hacker News (HN), a forum for general tech-centric discussions, and the /r/rust subreddit, a forum specifically about Rust. We selected these two forums both because they are frequent hosts to discussions about Rust and use upvote mechanisms to sort content. Upvotes act as a loose proxy for what readers consider to be good contributions to the conversation or perspectives they agree with, and have been shown to correlate with coarse notions of quality [11]. While upvote-based filtering can reduce exposure to controversial opinions, given the vast amount of possible content to read, we found it critical in improving the signal-to-noise ratio. To select the final corpus, we filtered for HN articles with "Rust" in the title, and considered all /r/rust articles. Then, we filtered for posts with at least 250/200 upvotes on HN/Reddit respectively, a total of 424 posts. From there, we selected posts that we felt were most relevant towards understanding user experiences in Rust, e.g. choosing "Three months of Rust" and "Why I'm dropping Rust' over "Announcing Rust 1.12". This ad hoc filter was not applied soundly or completely (we read as much as time permitted), so we do not claim our survey is exhaustive. Our final corpus contains 50 posts, with corresponding comment sections on both forums where applicable.

For each article, we performed a content analysis on both the document and at least the top five comments by upvotes for each forum the article appeared on. Then we categorized the articles (categories like "community", "ergonomics", "tooling", "security", etc.) and looked for trends within each category, forming preliminary hypotheses about barriers to adoption that were both novel and actionable to the Rust community. From the insights, we formed hypotheses that help explain the experiences that users encounter in our corpus. Concurrently, we interviewed three Rust language developers to help us contextualize our findings and to understand prevailing attitudes towards Rust's usability in the community.

3 Hypotheses

▶ **Hypothesis 1.** *Rust is primarily promoted for safety and speed; while those aspects matter to users, the tooling around Rust is equally valuable, but its value is not as clearly communicated by the language developers.*

Potential users need to understand Rust's features and goals in order to determine whether to use it. The authoritative source of information on Rust is its official website, `https://rust-lang.org`. On the front page and the FAQ, Rust is promoted as "a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety." Laundry lists of language features including "zero-cost abstractions," "pattern matching," and "type inference" are also provided. The standard library, the tooling, and the ecosystem are all absent from this messaging [1].

To evaluate whether Rust users found the language useful for its claimed benefits, we analyzed 12 experience reports (e.g. "Trying out Rust for Graphics Programming") and 6 language comparisons (e.g. "Comparing Rust and Java"). Across the 18 articles, we counted the reported pros/cons of Rust. The first and third most reported benefits of Rust were elimination of runtime errors (7 articles) and data races (5 articles). Runtime errors include both avoiding memory errors through Rust's memory safety analyzer as well as avoiding unhandled failures through sum types, e.g. `Option<T>`. This finding is consistent with Rust's messaging – users empirically self-report the utility of Rust's safety guarantees.

By contrast, the second most reported benefit of Rust was Cargo, Rust's build system and dependency manager (6 articles). One article summarized the collective sentiment [8]:

> Instead of having to invoke `pkg-config` by hand or with Autotools macros, wrangling include paths for header files and library files, and basically depending on the user to ensure that the correct versions of libraries are installed, you write a `Cargo.toml` file which lists the names and versions of your dependencies. [...] It just works when you `cargo build`.

The Rust language developers likely understand the importance of Rust's tooling, being a major part of the Rust 2017/18 roadmaps. However, because Rust's promotional messaging doesn't clearly emphasize these features, this absence suggests a possible disconnect between what the language developers and potential users consider the most important features of Rust.

▶ **Hypothesis 2.** *Complex pointer aliasing patterns are primarily implemented through existing libraries built on unsafe code, but Rust users have a hard time discovering these solutions.*

To guarantee memory safety for low-level programs with direct access to memory, Rust employs the "borrow checker", a static analysis tool that prevents memory-unsafe operations, e.g. returning a dangling pointer to a stack-allocated value. The borrow checker does not permit mutable aliases, or holding two mutable pointers to the same piece of memory. Aliasing patterns that violate this rule, like reference-counted pointers, can be implemented carefully through Rust's `unsafe` construct, often provided as libraries. However, this restriction is often daunting to Rust novices because it disallows patterns that are easy to express in other low-level languages, and it makes solutions difficult to find. Out of the 18 experience reports and language comparisons, the complexity of the borrow checker was the second most frequently mentioned complaint (only behind compiler version issues of stable vs. nightly). For example, one user implementing a video codec found [5]:

> Video codecs usually operate on planes and there you'd like to operate with different chunks of the frame buffer (or plane) at the same time. Rust does not allow you to mutably borrow parts of the same array even when it should be completely safe like `let mut a = &mut arr[0..pivot]; let mut b = &mut arr[pivot..];`.

Another user implementing a GUI framework found [3]:

> For `nanogui`... each widget has a pointer to a parent and a vector of pointers to its children. How does this concept map to Rust? There are several answers: 1. Use a naive `Vec<T>` implementation. 2. Use `Vec<*mut T>`. 3. Use `Vec<Rc<RefCell<T>>>`. 4. Use C bindings. ... I have tried options 1 through 3 with several drawbacks, each making them not fit for use. I'm currently looking at point 4 as my only remaining option to use.

In both cases, Rust users encountered a particular memory access pattern (disjoint mutable pointers to an array, widget trees with back pointers) that Rust disallowed. As with other complex aliasing patterns like reference counting, the solution to these types of problems is sufficiently complex that Rust users aren't expected to implement it themselves, but instead defer to external code. In both cases above, commenters pointed out standard library functions (`Vec::split_at_mut`) and third-party libraries (`petgraph`) for solving these issues respectively; however, the authors were not able to independently discover these solutions. These experiences suggest that Rust needs better resources to help users identify common aliasing patterns and understand what tools exist to solve those problems.

▶ **Hypothesis 3.** *Although incremental migration of existing codebases into Rust seems promising, Rust users aren't pursuing this path because the cost of integrating Rust into a different language ecosystem or toolchain is too great.*

One important path to adoption of Rust is incremental migration, or gradually rewriting components of a large software system from a host language (like C/C++) into Rust. Mozilla's initial motivation for Rust was to replace performance-critical parts of Firefox (Project Servo), and others have begun exploring integrating with databases (Postgres) and operating systems (Linux). However, the idea of incremental migration is almost entirely absent from the discourse we studied. Only one of our 50 articles directly described experiences integrating Rust into an existing system; most experience reports detailed new projects in Rust or entire rewrites of existing projects. While lack of discourse doesn't necessarily imply lack of adoption, it still does not bode well for the path of incremental migration – issues out of the public eye are unlikely to receive attention either from the Rust compiler developers in prioritizing work, or from the Rust community in generating documentation and tooling.

A possible explanation is that the challenges of incremental migration of Rust may be surmountable for larger teams, like those supporting Firefox, but are still too challenging for most Rust users. Discussion of incremental migration in our corpus largely occurred in scattered comments focused on describing challenges, not hailing great successes. For example, a Servo developer described in an interview the challenges of bridging Rust, C++, and Javascript in Firefox: the ease of accidentally invalidating a C++ reference when in Rust, the challenge of managing macros across the language barriers, and the complexity of tracking when each variable would be deallocated by which runtime [15].

Another example of a difficult challenge for most Rust users is working on Rust without Cargo, a tool which reduces the overhead of starting (and maintaining) a new project and reusing third-party Rust code. In low-level programming, build systems and package management are traditionally relegated to a hodgepodge of tools like Make, Autotools, CMake, and Apt; in contrast, the Rust community uses Cargo for both build process and dependency management. While Cargo is not strictly required, all major libraries must be built with Cargo. Developing Rust without Cargo means losing easy access to these libraries. For example, Facebook's Mononoke project, a rewrite of Mercurial in Rust, initially could

not use any packages outside the standard library due to integration requirements with Facebook's custom build system, which significantly slowed adoption of external libraries. The lack of discourse around these issues suggest more resources should potentially be dedicated to reducing these barriers to adoption for incremental migration into Rust.

4 Discussion

By applying an ethnomethodological approach to Rust's online discourse, we identified supporting evidence for three hypotheses about factors which meaningfully influence the adoption of Rust. This methodology is neither fully precise nor conclusive, but it provides a useful signal to direct further evaluation of these hypotheses. While online commenters are not subject to the same level of rigor as peer-reviewed research, our experience suggests that blog posts and forum threads still contain an enormous amount of collective wisdom that is perhaps under-appreciated in academic literature. Much of the prior work has trended towards precision through controlled experiments or breadth through surveys and code base analyses. However, we believe that understanding these online communities can provide valuable guidance to PL/HCI researchers seeking to address the key problems facing today's programmers.

To that end, one key question is this study's replicability: could other researchers recreate our results for Rust, or perform the same study on other programming languages? Due to the time-consuming nature of manual content analysis, consistently filtering articles from the larger corpus is important. Creating better filters would be simplified by looking at a smaller domain of documents with more quantifiable filters, e.g. analyzing just experience reports, or articles about security, or articles about language design decisions. Conversely, studies that attempt to just categorize the discourse by identifying common topics can reduce work done in later studies on any given category. Lastly, applying a consistent content analysis methodology is challenging given the open-ended, free-form nature of synthesizing connections in document surveys. Perhaps here the PL/HCI community could provide more guidance on standard methodologies for ethnomethodological analyses.

The second question is the applicability of this study's results: what further experiments do our results suggest? We believe the logical next step is to develop surveys targeted towards understanding the extent of the issues identified above. While usability surveys often resort to generic questions like "where do you think Rust can improve?" (e.g. as in [13]), the hypotheses above suggest more specific questions like:

- How has your perception of Cargo's value changed between starting with Rust and today?
- What parts of Rust do you find most useful that you weren't told about initially?
- What are examples of times when you couldn't solve an issue with the borrow checker after several hours? If you eventually solved it, how did you solve it?
- What factors influence your likelihood to adopt Rust into an existing non-Rust project?

Given feedback on these questions from the community, future work could explore controlled experiments for more replicable identification of usability issues, as well as analyze how insights gleaned from Rust can generalize to other programming language communities.

References

1 The Rust Programming Language. URL: `https://www.rust-lang.org/en-US/`.

2 Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 332–343. IEEE, 2016.

3 Michael de Lang. Why I'm dropping Rust, September 2016. URL: https://hackernoon.com/why-im-dropping-rust-fd1c32986c88.
4 Harold Garfinkel. *Studies in ethnomethodology*. Prentice-Hall, 1967.
5 Kotsya. Rust: Not So Great for Codec Implementing, July 2017. URL: https://codecs.multimedia.cx/2017/07/rust-not-so-great-for-codec-implementing/.
6 Manishearth. Rust severely disappoints me. Reddit, January 2017. URL: https://www.reddit.com/r/rust/comments/5nl3fk/rust_severely_disappoints_me/.
7 David Martin, Benjamin V Hanrahan, Jacki O'Neill, and Neha Gupta. Being a turker. In *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*, pages 224–235. ACM, 2014.
8 Federico Mena-Quintero. Rust things I miss in C, February 2018. URL: https://people.gnome.org/~federico/blog/rust-things-i-miss-in-c.html.
9 Leo A Meyerovich and Ariel S Rabkin. Empirical analysis of programming language adoption. *ACM SIGPLAN Notices*, 48(10):1–18, 2013.
10 Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.
11 Greg Stoddard. Popularity and quality in social news aggregators: A study of reddit and hacker news. In *Proceedings of the 24th international conference on world wide web*, pages 815–818. ACM, 2015.
12 The Rust Core Team. Rust's 2018 roadmap, March 2018. URL: https://blog.rust-lang.org/2018/03/12/roadmap.html.
13 Jonathan Turner. Rust 2017 Survey Results, September 2017. URL: https://blog.rust-lang.org/2017/09/05/Rust-2017-Survey-Results.html.
14 Anna Zeng, Will Crichton, and Niko Matsakis. Interview with Niko Matsakis, May 2018.
15 Anna Zeng and Josh Matthews. Interview with Josh Matthews, May 2018.

Observing the Uptake of a Language Change Making Strings Immutable

Manuel Maarek
Heriot-Watt University, Edinburgh, Scotland, UK
m.maarek@hw.ac.uk
https://orcid.org/0000-0001-6233-6341

Abstract

To address security concerns, a major change was introduced to the OCaml language and compiler which made strings immutable and introduced array of bytes as replacement for mutable strings. The change is progressively being pushed so that ultimately strings will be immutable. We have investigated the way OCaml package developers undertook the change. In this paper we report on a preliminary observation of software code from the main OCaml package management system. For this purpose we instrumented versions of the OCaml compiler to get precise information into the uptake of safe strings.

2012 ACM Subject Classification Software and its engineering → Software evolution

Keywords and phrases software evolution, programming language evaluation, immutability, secure programming

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2018.6

1 Introduction

Following the LaFoSec study [6], a major change was introduced to the OCaml language and compiler which splits the `string` data type into its now immutable version and a new `bytes` data type representing arrays of bytes. The change is gradual and was initially made in version 4.02.0 (see Table 1). The new immutable strings were available in the language and standard library as well as the new array of bytes but the compiler would initially still allow to mutate strings, issuing a warning to the developer. Immutable strings are now default unless a specific option is set. In future versions of the compiler strings will only be immutable.

The security rationale for the change was that any part of a code could alter a string value it has access to regardless of encapsulation (e.g. string literals of the standard library could be modified). The importance of the change and the fact the change is being carried out gradually to retain backward compatibility during a period of time makes it an interesting case to evaluate how developers perceive such changes and how they implement it within their own development. As a starting point, we want to mechanically inspect the uptake within developers' codes. We have therefor developed versions of the compiler that observe the uses of `string`, `bytes`, their associated operations, and the relevant compiler options. We then run our observation compilers on openly accessible OCaml code. We chose to mine the OCaml code that are provided through OCaml's main package management system OPAM. In this paper, we present the experiment setting and discuss our findings.

Table 1 OCaml versions significant for string immutability.

	OCaml version	OPAM version	Number of packages
Mutable strings	**4.01.0** (2013-09)	1.1.2 (2014-06)	1382
Introduction of safe strings	4.02.0 (2014-08)		
Last version of the 4.02 family	**4.02.3** (2015-08)	1.2.1 (2015-03)	1699
Safe string as default	4.06.0 (2017-11)		
Latest version	**4.07.0** (2018-07)	1.2.2 (2015-04)	1921

Plan

In Section 2 we give more details about immutability, OCaml and LaFoSec which form the context of this research. We present our experimental setting in Section 3 and the outcome of our observations in Section 4. We finally conclude and discuss perspectives for this work in Section 5.

2 Context

In this section we give the context to this research where we planned an observation of the uptake of a security-induced change in the OCaml language.

2.1 Immutability

Mutability is a key feature of most programming language as it is a way for programmer to represent the change of state of the system they develop. Some programming languages such as functional languages prevent variable mutation, the state of the system is represented by the arguments passed from operations to operations following the execution flows of the program. An advantage of immutability is that it enables referential transparency which means that a variable will always represent the same value within its scope: passing the variable as argument to other operations or concurrent operations cannot modify its value. As a result immutability is more secure as it brings certainty to the programmer. Mutability is a source of vulnerability as the following CWEs (Common Weakness Enumeration) exemplify: *CWE-374: Passing Mutable Objects to an Untrusted Method*[1], *CWE-471: Modification of Assumed-Immutable Data (MAID)*[2]. Immutability is increasingly available in programming languages although immutability could cause usability challenges for developers used to state-base programming paradigms [11, 3].

2.2 OCaml

OCaml[3] is a functional language. While being a functional language, it offers some mutable data structures and variable references. In OCaml, strings are mutable since the early versions of the language. This design choice seems to have been made to allow for easy use of strings as byte arrays where mutability is essential.

[1] `https://cwe.mitre.org/data/definitions/374.html`
[2] `https://cwe.mitre.org/data/definitions/471.html`
[3] `https://ocaml.org/`

OCaml is also a strongly typed programming which means that each expression is associated to a unique type. Its inference system implemented with its type system in the compiler, makes it unnecessary in most cases to annotate the source code with type information.

2.3 LaFoSec study

The LaFoSec study [6] analysed the intrinsic security properties of functional languages and OCaml in particular. It was preceded by a study on Java Security [5]. LaFoSec detailed the features such as encapsulation which are essential for security development, and identified ways in which they could be bypassed. It highlighted the security issue of immutable strings. Let us consider the following example of a function returning the days of the week as strings (using OCaml 4.01.0 version).

```
# let weekday n =
      match n with
      | 1 -> "Monday"
      | ...
val weekday : int -> string = <fun>
# weekday 1
- : string = "Monday"
```

The string literal of the function could be changed by any piece of code executed outside of the current module, e.g. by any code dynamically loaded.

```
# let s = weekday 1; s.[0] <- 'F'; s.[1] <- 'r'; s.[2] <- 'i';
# weekday 1
- : string = "Friday"
```

To prevent such tempering, LaFoSec recommended to protect string literals by wrapping them in a call to the `String.copy` function, which means that each call to `weekday` will issue a new copy of the string literal. In addition, LaFoSec recommended to define an abstract module wrapping strings and which only offers access operations that do not mutate strings. These solutions were implemented in the secure XML validator prototype developed as part of LaFoSec and presented in [4]. The issue the mutable strings could cause to the reliability of higher level formal development is discussed in [1], highlighting that one can temper with the output of a logical frameworks. And [2] is another example of a formal development which required to rely on a custom string library for the system to retain security guarantees.

2.4 Change

The change introduced by the OCaml development team makes the values of the type `string` immutable by default and introduce a new type `bytes` for byte arrays (distinguishing the two common uses of strings). The change could imply additional operations and therefor a performance cost when needing to use `string` operations on `bytes` values as discussed in [10].

The change means that the methods of the `String` module (of the standard library) which were modifying strings in-place are now deprecated. When used, they raise a warning or an error if the `-safe-string` option of the compiler is used. Here is an example of warning issued by the 4.07.0 compiler when calling `String.set`.

```
# String.set;;
Warning 3: deprecated: Stdlib.String.set
Use Bytes.set instead.
- : bytes -> int -> char -> unit = <fun>
```

Note that the `String.set` function, which remained for backward compatibility, is an now alias to `Bytes.set` and expect as first argument a value of type bytes unless in unsafe mode. Calling `String.set` with a value of type `string` raises by default a type error with OCaml 4.07.0 version (OCaml 4.02.3 version would allow it by default).

```
# String.set "Monday";;
Warning 3: deprecated: Stdlib.String.set
Use Bytes.set instead.
Error: This expression has type string but an expression was expected
         of type bytes
```

The functions that are now deprecated following the change are `String.set` (which corresponds to the `s.[0] <- 'F'` syntax sugaring notation), `String.create`, `String.copy`, `String.fill`. Note that `unsafe_*` variants of `String` methods existed and are still available. E.g., `String.unsafe_get` allows to access a character of a string by its index as `String.get` does, but does it without runtime verification that the access is done within the boundaries of the string (this is used for optimisation purposes).

3 Experimental Setup

Instrumented compilers

To observe the manner OCaml users have adopted the new immutable strings and its mutable counterpart, we created variants of the OCaml compiler that records specific usage information. These instrumented compilers record the compiling options, the occurrence of expressions of type `string` and `bytes`, and the occurrence of calls to string and bytes methods[4].

Working within the compiler means that we have certainty about the information we gather: type information has been inferred and checked by the compiler. An alternative would have been to work at the source level where similar content could have different forms (e.g. qualified forms vs. unqualified forms), where some information are not available (e.g. type information, command line options), and noise needs to be filtered out (e.g. commented code).

We developed three instrumented compilers (or observer compilers) for the OCaml versions we listed in Table 1. We refer to the compiler for OCaml 4.01.0 version as `4.01.0` (resp. `4.02.3`, `4.07.0`) and to our instrumented compiler as `4.01.0+obs` (resp. `4.02.3+obs`, `4.07.0+obs`). Note that the three versions of OCaml are not retrocompatible and therefore require dedicated compilers.

OCaml being a compiled and strongly typed language, a compilation is necessary to execute OCaml programs and the compilation process includes typing the program after parsing and before code generation. The instrumentation we created are placed at the end of the typing when a typed Abstract Syntax Tree (AST) of the program is available.

Note that our current instrumented compilers look for `string`/`bytes` expressions and `string`/`bytes` methods without investing further the context of use or within the program flows. We are considering making such finer grained investigation in later research.

[4] Note that some of the information we collected could be obtained with the `-annot` option but would have required additional processing.

Code repository

We also decided to work within OCaml's OPAM package management system[5] as source codes in such system come alongside compiling instructions. This means that we were able to automate the process of gathering information about individual packages and files. While using OPAM means that we could easily access compilation commands, it also implies that the targeted audience are package developers who are more likely to be professionals or experts rather than general users of OCaml.

OPAM makes it trivial to deploy such custom compiler and retaining the dependency of packages of the original version.

Related work on code mining

A number of related works have been mining openly available code [9], or have investigated version histories to observe change [12, 8], sometimes working on the AST rather than source code [7].

4 Results of Observations

We identified packages that were available for each of the three versions of OCaml we wanted to base our research on (see Table 1) highlighting stages of the gradual change. The number of commonly available packages is 353. We have therefore used our observation versions of the compilers to install these 353 packages as we were interested in seeing the changes developer made since the introduction of immutable strings. Although we installed the same packages for each compiler version, or more precisely the versions of the same packages that corresponded to each compiler version, the number of files that were compiled successfully varied depending of the version. Using the default compilers (`4.01.0`, `4.02.3`, `4.07.0`), some compilations failed. This was in particular the case for `4.02.3` which could be explained by the fact that 4.02 family saw a number of minor versions making packages maintenance less effective. A package failing to compile meant that others could not be compiled due to missing dependency. A small number of compilations which compiled successfully with the default compiler failed or stalled with our instrumented compilers (`4.01.0+obs`, `4.02.3+obs`, `4.07.0+obs`). This was due to the instrumentation we designed with a compiler module to iterate over the OCaml AST which would sometimes reach the stack limit. As Table 2 shows, less files compiled successfully with version 4.02.3.

Table 2 Number of files compiled per version.

Version	Number of packages installed successfully	Number of files compiled successfully
`4.01.0`	212	
`4.01.0+obs`	201	7211
`4.02.3`	46	
`4.02.3+obs`	38	2253
`4.07.0`	224	
`4.07.0+obs`	221	10032

[5] `https://opam.ocaml.org/`

Use of unsafe string compiler option

We tracked the use of compiler options. This is only applicable to versions of the compiler released after the change. The gradual transition meant that there was limited incentive for the developers to use the safe compilation option in 4.02.3 version while 4.07.0 version does. Table 3 shows that such change of behaviour by the compiler were followed by an uptake of safer compiler options.

Table 3 Use of unsafe compiler option.

Version	Package use of unsafe option	File compiled with unsafe option
`4.01.0+obs`	NA / 201	NA / 7211
`4.02.3+obs`	30 / 38	2100 / 2253
`4.07.0+obs`	0 / 221	0 / 10032

Expressions of type `string` and `bytes`

The observation compilers also counted the number of expressions of type `string` and `bytes` as well as the overall number of expressions in the files that compiled successfully. This information gives an estimate of the use of these types of expression although it did not include functions with these types as either argument or output. Table 4 shows a slight decrease of the number of `bytes` expressions in proportion to `string` expressions.

Table 4 Expressions of type `string`/`bytes`.

Version	`string` expressions	`bytes` expressions	Ratio	Total number of expressions
`4.01.0+obs`	732390	NA	NA	7332863
`4.02.3+obs`	205165	3335	0.0162	2197761
`4.07.0+obs`	864876	13466	0.0155	9989802

Calls to unsafe `String` functions

The observation compilers record the usage of unsafe functions from the `String` module of the standard library. This recording is not transitive, so if a call is made to a function which is an alias to a unsafe `String` function, this will not be identified. Table 5 shows a decrease in the number of calls to unsafe `String` functions, although for 4.02.3 version the decrease could also be due to the set of packages that compiled rather than being due to the introduction of the change.

Table 5 Call to unsafe `String` functions.

Version	Packages with call to unsafe functions	Files with call to unsafe functions
`4.01.0+obs`	43 / 201	153 / 7211 (2.12%)
`4.02.3+obs`	5 / 38	16 / 2253 (0.71%)
`4.07.0+obs`	2 / 221	4 / 10032 (0.03%)

Note that the few remaining cases of call to unsafe `String` methods with 4.07.0 version are the consequence of syntax sugaring. The syntax sugaring notations to get (`s.[2]`) or set

(`s.[2] <- 'i'`) a character are respectively translated into the following calls `String.get s` and `String.set s 2 'i'`. This is still the case for 4.07.0 version which results in the type checker issuing a warning when identifying the `String.set` method and accepting its call when the argument is of type `bytes` (see an illustration with the following two snippets of code). This explains why for 4.07.0 version, the cases of call to `String.set` are compatible with the the absence of use of unsafe string compiler option.

```
# let s = "Monday";;
val s : string = "Monday"
# s.[2] <- 'i';;
Warning 3: deprecated: Stdlib.String.set
Use Bytes.set instead.
Error: This expression has type string but an expression was expected
         of type bytes
```

```
# let b = Bytes.of_string "Monday";;
val b : bytes = Bytes.of_string "Monday"
# b.[2] <- 'i';;
Warning 3: deprecated: Stdlib.String.set
Use Bytes.set instead.
- : unit = ()
```

5 Conclusion and Future Work

The purpose of our research is to understand how software developers refactor their code in light of security focused language changes. In this paper we presented our compiler instrumentation and initial findings into the uptake of a gradual change introduced in the OCaml language and compiler to make strings immutable. We relied on the OCaml type system and package management system to create instrumented compilers collecting accurate information about the uptake. Although the study only looked at three versions of the compiler, it shows a good uptake by the OCaml package developers.

As this is an initial experiment there are many ways it could be improved and expanded. We aim to apply the same strategy to more versions of the compiler to get a clearer picture of the uptake. We also would like to expand the investigation by looking at more diverse repositories of code, and comparing with similar changes in other programming languages.

We have not investigate in details how individual development proceeded with their change and would like to do so to categories these strategies. It would be interesting to combine this code based investigation with the way the change and strategy were perceived by OCaml developers. This could be the base to tailor such changes to optimise uptake or to create adequate interventions to drive changes.

References

1 Mark Adams. Flyspecking Flyspeck. In *Mathematical Software – ICMS 2014*, Lecture Notes in Computer Science, pages 16–20. Springer, Berlin, Heidelberg, August 2014. doi:10.1007/978-3-662-44199-2_3.

2 David Cadé and Bruno Blanchet. Proved Generation of Implementations from Computationally Secure Protocol Specifications1. *Journal of Computer Security*, 23(3):331–402, January 2015. doi:10.3233/JCS-150524.

3 M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine. Glacier: Transitive Class Immutability for Java. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 496–506, May 2017. doi:10.1109/ICSE.2017.52.

4 D. Doligez, C. Faure, T. Hardin, and M. Maarek. Avoiding Security Pitfalls with Functional Programming: A Report on the Development of a Secure XML Validator. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 209–218, May 2015. doi:10.1109/ICSE.2015.149.

5 Éric Jaeger, Olivier Levillain, and Pierre Chifflier. Mind Your Language (s) – A Discussion about Languages and Security (Long Version). In *First Workshop on Language-Theoretic Security (LangSec) at the IEEE CS Security & Privacy Workshops*, 2014.

6 LaFoSec. Security and Functional Languages (Étude de La Sécurité Intrinsèque Des Langages Fonctionnels). Technical report, ANSSI (National Cybersecurity Agency of France), Main authors: D. Doligez, C. Faure, T. Hardin, M. Maarek, 2011.

7 M. Martinez, L. Duchien, and M. Monperrus. Automatically Extracting Instances of Code Change Patterns with AST Analysis. In *2013 IEEE International Conference on Software Maintenance*, pages 388–391, September 2013. doi:10.1109/ICSM.2013.54.

8 O. Meqdadi, N. Alhindawi, M. L. Collard, and J. I. Maletic. Towards Understanding Large-Scale Adaptive Changes from Version Histories. In *2013 IEEE International Conference on Software Maintenance*, pages 416–419, September 2013. doi:10.1109/ICSM.2013.61.

9 Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165, New York, NY, USA, 2014. ACM. doi:10.1145/2635868.2635922.

10 Gerd Stolpmann. Immutable Strings in OCaml-4.02 (Blog on Camlcity.Org). http://blog.camlcity.org/blog/bytes1.html, July 2014.

11 S. Weber, M. Coblenz, B. Myers, J. Aldrich, and J. Sunshine. Empirical Studies on the Security and Usability Impact of Immutability. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 50–53, September 2017. doi:10.1109/SecDev.2017.21.

12 T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005. doi:10.1109/TSE.2005.72.